nipype Documentation

Release 0.11.0

Neuroimaging in Python team

September 15, 2015, 17:26 PDT

Contents

Previous versions: 0.10.0 0.9.2

Guides

- User

CHAPTER 1

User Guide

Release 0.11.0
Date September 15, 2015, 17:26 PDT

1.1 Download and install

This page covers the necessary steps to install Nipype.

1.1.1 Download

Release 0.10.0: [zip tar.gz]
Development: [zip tar.gz]
Prior downloads
To check out the latest development version:

```
git clone git://github.com/nipy/nipype.git
```

1.1.2 Install

The installation process is similar to other Python packages.
If you already have a Python environment setup that has the dependencies listed below, you can do:

```
easy_install nipype
```

or:

```
pip install nipype
```

Debian and Ubuntu

Add the NeuroDebian repository and install the `python-nipype` package using `apt-get` or your favorite package manager.

Mac OS X

The easiest way to get nipype running on Mac OS X is to install Anaconda or Canopy and then add nibabel and nipype by executing:

```
easy_install nibabel
easy_install nipype
```

From source

If you downloaded the source distribution named something like `nipype-x.y.tar.gz`, then unpack the tarball, change into the `nipype-x.y` directory and install nipype using:

```
python setup.py install
```

Note: Depending on permissions you may need to use `sudo`.

1.1.3 Testing the install

The best way to test the install is to run the test suite. If you have nose installed, then do the following:

```
python -c "import nipype; nipype.test()"
```

you can also test with nosetests:

```
nosetests --with-doctest /software/nipy-repo/masternipype/nipype
--exclude=external --exclude=testing
```

All tests should pass (unless you're missing a dependency). If SUBJECTS_DIR variable is not set some FreeSurfer related tests will fail. If any tests fail, please report them on our bug tracker.
On Debian systems, set the following environment variable before running tests:

```
export MATLABCMD=$pathtomatlabdir/bin/$platform/MATLAB
```

where, $pathtomatlabdir is the path to your matlab installation and $platform is the directory referring to x86 or x64 installations (typically glnxa64 on 64-bit installations).

Avoiding any MATLAB calls from testing

On unix systems, set an empty environment variable:

```
export NIPYPE_NO_MATLAB=
```

This will skip any tests that require matlab.

1.1.4 Dependencies

Below is a list of required dependencies, along with additional software recommendations.

Must Have

Python 2.7
Nibabel 1.0 - 1.4 Neuroimaging file i/o library
NetworkX 1.0 - 1.8 Python package for working with complex networks.
NumPy 1.3 - 1.7
SciPy 0.7 - 0.12 Numpy and Scipy are high-level, optimized scientific computing libraries.
Enthought Traits 4.0.0 - 4.3.0
Dateutil 1.5 -

Note: Full distributions such as Anaconda or Canopy provide the above packages, except Nibabel.

Strong Recommendations

IPython 0.10.2 - 1.0.0 Interactive python environment. This is necessary for some parallel components of the pipeline engine.
Matplotlib 1.0 - 1.2 Plotting library
RDFLib 4.1 RDFLibrary required for provenance export as RDF
Sphinx 1.1 Required for building the documentation
Graphviz Required for building the documentation

Interface Dependencies

These are the software packages that nipype.interfaces wraps:

FSL 4.1.0 or later
matlab 2008a or later
SPM SPM5/8
FreeSurfer FreeSurfer version 4 and higher
AFNI 2009_12_31_1431 or later
Slicer 3.6 or later
Nipy 0.1.2+20110404 or later
Nitime (optional)
Camino
Camino2Trackvis
ConnectomeViewer

1.2 Running Nipype in a VM

Tip: Creating the Vagrant VM as described below requires an active internet connection.

Container technologies (Vagrant, Docker) allow creating and manipulating lightweight virtual environments. The Nipype source now contains a Vagrantfile to launch a Vagrant VM.
Requirements:

- Vagrant
- Virtualbox

After you have installed Vagrant and Virtualbox, you simply need to download the latest Nipype source and unzip/tar/compress it. Go into your terminal and switch to the nipype source directory. Make sure the Vagrantfile is in the directory. Now you can execute:

```
vagrant up
```

This will launch and provision the virtual machine.
The default virtual machine is built using Ubuntu Precise 64, linked to the NeuroDebian source repo and contains a 2 node Grid Engine for cluster execution.
The machine has a default IP address of *192.168.100.20* . From the vagrant startup directory you can log into the machine using:

```
vagrant ssh
```

Now you can install your favorite software using:

```
sudo apt-get install fsl afni
```

Also note that the directory in which you call *vagrant up* will be mounted under */vagrant* inside the virtual machine. You can also copy the Vagrantfile or modify it in order to mount a different directory/directories.
Please read through Vagrant documentation on other features. The python environment is built using a miniconda distribution. Hence *conda* can be used to do your python package management inside the VM.

1.3 Tutorial : Interfaces

1.3.1 Specifying options

The nipype interface modules provide a Python interface to external packages like FSL and SPM. Within the module are a series of Python classes which wrap specific package functionality. For example, in the fsl module, the class `nipype.interfaces.fsl.Bet` wraps the `bet` command-line tool. Using the command-line tool, one would specify options using flags like `-o`, `-m`, `-f <f>`, etc... However, in nipype, options are assigned to Python attributes and can be specified in the following ways:
Options can be assigned when you first create an interface object:

```
import nipype.interfaces.fsl as fsl
mybet = fsl.BET(in_file='foo.nii', out_file='bar.nii')
result = mybet.run()
```

Options can be assigned through the `inputs` attribute:

```
import nipype.interfaces.fsl as fsl
mybet = fsl.BET()
mybet.inputs.in_file = 'foo.nii'
mybet.inputs.out_file = 'bar.nii'
result = mybet.run()
```

Options can be assigned when calling the `run` method:

```
import nipype.interfaces.fsl as fsl
mybet = fsl.BET()
result = mybet.run(in_file='foo.nii', out_file='bar.nii', frac=0.5)
```

1.3.2 Getting Help

In IPython you can view the docstrings which provide some basic documentation and examples.

```
In [2]: fsl.FAST?
Type:               type
Base Class: <type 'type'>
String Form:        <class 'nipype.interfaces.fsl.preprocess.FAST'>
Namespace:  Interactive
File:               /Users/satra/sp/nipype/interfaces/fsl/preprocess.py
Docstring:
    Use FSL FAST for segmenting and bias correction.

    For complete details, see the `FAST Documentation.
    <http://www.fmrib.ox.ac.uk/fsl/fast4/index.html>`_

    Examples
    --------
    >>> from nipype.interfaces import fsl
    >>> from nipype.testing import anatfile

    Assign options through the ``inputs`` attribute:

    >>> fastr = fsl.FAST()
    >>> fastr.inputs.in_files = anatfile
    >>> out = fastr.run() #doctest: +SKIP

Constructor information:
Definition: fsl.FAST(self, **inputs)
```

```
In [5]: spm.Realign?
Type:               type
Base Class:  <type 'type'>
String Form: <class 'nipype.interfaces.spm.preprocess.Realign'>
Namespace:   Interactive
File:               /Users/satra/sp/nipype/interfaces/spm/preprocess.py
Docstring:
    Use spm_realign for estimating within modality rigid body alignment

    http://www.fil.ion.ucl.ac.uk/spm/doc/manual.pdf#page=25

    Examples
```

```
        --------

        >>> import nipype.interfaces.spm as spm
        >>> realign = spm.Realign()
        >>> realign.inputs.in_files = 'functional.nii'
        >>> realign.inputs.register_to_mean = True
        >>> realign.run() # doctest: +SKIP

    Constructor information:
    Definition:  spm.Realign(self, **inputs)
```

All of the nipype.interfaces classes have an `help` method which provides information on each of the options one can assign.

```
In [6]: fsl.BET.help()
Inputs
------

Mandatory:
 in_file: input file to skull strip

Optional:
 args: Additional parameters to the command
 center: center of gravity in voxels
 environ: Environment variables (default={})
 frac: fractional intensity threshold
 functional: apply to 4D fMRI data
  mutually exclusive: functional, reduce_bias
 mask: create binary mask image
 mesh: generate a vtk mesh brain surface
 no_output: Don't generate segmented output
 out_file: name of output skull stripped image
 outline: create surface outline image
 output_type: FSL output type
 radius: head radius
 reduce_bias: bias field and neck cleanup
  mutually exclusive: functional, reduce_bias
 skull: create skull image
 threshold: apply thresholding to segmented brain image and mask
 vertical_gradient: vertical gradient in fractional intensity threshold (-1, 1)

Outputs
-------
mask_file: path/name of binary brain mask (if generated)
meshfile: path/name of vtk mesh file (if generated)
out_file: path/name of skullstripped file
outline_file: path/name of outline file (if generated)
```

```
In [7]: spm.Realign.help()
Inputs
------

Mandatory:
 in_files: list of filenames to realign

Optional:
 fwhm: gaussian smoothing kernel width
 interp: degree of b-spline used for interpolation
 jobtype: one of: estimate, write, estwrite (default=estwrite)
```

```
     matlab_cmd: None
     mfile: Run m-code using m-file (default=True)
     paths: Paths to add to matlabpath
     quality: 0.1 = fast, 1.0 = precise
     register_to_mean: Indicate whether realignment is done to the mean image
     separation: sampling separation in mm
     weight_img: filename of weighting image
     wrap: Check if interpolation should wrap in [x,y,z]
     write_interp: degree of b-spline used for interpolation
     write_mask: True/False mask output image
     write_which: determines which images to reslice
     write_wrap: Check if interpolation should wrap in [x,y,z]

    Outputs
    -------
    mean_image: Mean image file from the realignment
    realigned_files: Realigned files
    realignment_parameters: Estimated translation and rotation parameters
```

Our interface-index documentation provides html versions of our docstrings and includes links to the specific package documentation. For instance, the `nipype.interfaces.fsl.Bet` docstring has a direct link to the online BET Documentation.

1.3.3 FSL interface example

Using FSL to realign a time_series:

```
import nipype.interfaces.fsl as fsl
realigner = fsl.McFlirt()
realigner.inputs.in_file='timeseries4D.nii'
result = realigner.run()
```

1.3.4 SPM interface example

Using SPM to realign a time-series:

```
import nipype.interfaces.spm as spm
from glob import glob
allepi = glob('epi*.nii') # this will return an unsorted list
allepi.sort()
realigner = spm.Realign()
realigner.inputs.in_files = allepi
result = realigner.run()
```

1.4 Interface caching

This section details the interface-caching mechanism, exposed in the `nipype.caching` module.

1.4.1 Interface caching: why and how

- *Pipelines* (also called *workflows*) specify processing by an execution graph. This is useful because it opens the door to dependency checking and enable *i)* to minimize recomputations, *ii)* to have the execution engine transparently deal with intermediate file manipulations.
  They however do not blend in well with arbitrary Python code, as they must rely on their own execution engine.
- *Interfaces* give fine control of the execution of each step with a thin wrapper on the underlying software. As a result that can easily be inserted in Python code.

However, they force the user to specify explicit input and output file names and cannot do any caching.
This is why nipype exposes an intermediate mechanism, *caching* that provides transparent output file management and caching within imperative Python code rather than a workflow.

1.4.2 A big picture view: using the `Memory` object

nipype caching relies on the *Memory* class: it creates an execution context that is bound to a disk cache:

```
>>> from nipype.caching import Memory
>>> mem = Memory(base_dir='.')
```

Note that the caching directory is a subdirectory called *nipype_mem* of the given *base_dir*. This is done to avoid polluting the base director.
In the corresponding execution context, nipype interfaces can be turned into callables that can be used as functions using the *Memory.cache()* method. For instance if we want to run the fslMerge command on a set of files:

```
>>> from nipype.interface import fsl
>>> fsl_merge = mem.cache(fsl.Merge)
```

Note that the *Memory.cache()* method takes interfaces **classes**, and not instances.
The resulting *fsl_merge* object can be applied as a function to parameters, that will form the inputs of the *merge* fsl commands. Those inputs are given as keyword arguments, bearing the same name as the name in the inputs specs of the interface. In IPython, you can also get the argument list by using the *fsl_merge?* synthax to inspect the docs:

```
In [10]: fsl_merge?
String Form:PipeFunc(nipype.interfaces.fsl.utils.Merge, base_dir=/home/varoquau/dev/nipype/nipyp
Namespace:  Interactive
File:       /home/varoquau/dev/nipype/nipype/caching/memory.py
Definition: fsl_merge(self, **kwargs)
Docstring:
Use fslmerge to concatenate images

Inputs
------

Mandatory:
dimension: dimension along which the file will be merged
in_files: None

Optional:
args: Additional parameters to the command
environ: Environment variables (default={})
ignore_exception: Print an error message instead of throwing an exception in case the interface
merged_file: None
output_type: FSL output type

Outputs
-------
merged_file: None
Class Docstring:
...
```

Thus *fsl_merge* is applied to parameters as such:

```
>>> results = fsl_merge(dimension='t', in_files=['a.nii.gz', 'b.nii.gz'])
INFO:workflow:Executing node faa7888f5955c961e5c6aa70cbd5c807 in dir: /home/varoquau/dev/nipype/
INFO:workflow:Running: fslmerge -t /home/varoquau/dev/nipype/nipype/caching/nipype_mem/nipype-in
```

The results are standard nipype nodes results. In particular, they expose an *outputs* attribute that carries all the

outputs of the process, as specified by the docs.

```
>>> results.outputs.merged_file
'/home/varoquau/dev/nipype/nipype/caching/nipype_mem/nipype-interfaces-fsl-utils-Merge/faa7888f5
```

Finally, and most important, if the node is applied to the same input parameters, it is not computed, and the results are reloaded from the disk:

```
>>> results = fsl_merge(dimension='t', in_files=['a.nii.gz', 'b.nii.gz'])
INFO:workflow:Executing node faa7888f5955c961e5c6aa70cbd5c807 in dir: /home/varoquau/dev/nipype/
INFO:workflow:Collecting precomputed outputs
```

Once the *Memory* is set up and you are applying it to data, an important thing to keep in mind is that you are using up disk cache. It might be useful to clean it using the methods that *Memory* provides for this: *Memory.clear_previous_runs()*, *Memory.clear_runs_since()*.

Example

A full-blown example showing how to stage multiple operations can be found in the `caching_example.py` file.

1.4.3 Usage patterns: working efficiently with caching

The goal of the *caching* module is to enable writing plain Python code rather than workflows. Use it: instead of data grabber nodes, use for instance the *glob* module. To vary parameters, use *for* loops. To make reusable code, write Python functions.
One good rule of thumb to respect is to avoid the usage of explicit filenames apart from the outermost inputs and outputs of your processing. The reason being that the caching mechanism of `nipy.caching` takes care of generating the unique hashes, ensuring that, when you vary parameters, files are not overridden by the output of different computations.

Debuging

If you need to inspect the running environment of the nodes, it may be useful to know where they were executed. With *nipype.caching*, you do not control this location as it is encoded by hashes.
To find out where an operation has been persisted, simply look in it's output variable:

```
out.runtime.cwd
```

Finally, the more you explore different parameters, the more you risk creating cached results that will never be reused. Keep in mind that it may be useful to flush the cache using *Memory.clear_previous_runs()* or *Memory.clear_runs_since()*.

1.4.4 API reference

The main class of the `nipype.caching` module is the *Memory* class:

class `nipype.caching.`**Memory**(*base_dir*)

Memory context to provide caching for interfaces

Parameters **base_dir: string** :

The directory name of the location for the caching

Methods

---

cache(*interface*)

Returns a callable that caches the output of an interface

Parameters **interface: nipype interface** :
The nipype interface class to be wrapped and cached
Returns **pipe_func: a PipeFunc callable object** :
An object that can be used as a function to apply the interface to arguments. Inputs of the interface are given as keyword arguments, bearing the same name as the name in the inputs specs of the interface.

Examples

```
>>> from tempfile import mkdtemp
>>> mem = Memory(mkdtemp())
>>> from nipype.interfaces import fsl
```

Here we create a callable that can be used to apply an fsl.Merge interface to files

```
>>> fsl_merge = mem.cache(fsl.Merge)
```

Now we apply it to a list of files. We need to specify the list of input files and the dimension along which the files should be merged.

```
>>> results = fsl_merge(in_files=['a.nii', 'b.nii'],
...                     dimension='t')
```

We can retrieve the resulting file from the outputs: >>> results.outputs.merged_file # doctest: +SKIP '...'

clear_previous_runs(*warn=True*)
Remove all the cache that where not used in the latest run of the memory object: i.e. since the corresponding Python object was created.
Parameters **warn: boolean, optional** :
If true, echoes warning messages for all directory removed

clear_runs_since(*day=None*, *month=None*, *year=None*, *warn=True*)
Remove all the cache that where not used since the given date
Parameters **day, month, year: integers, optional** :
The integers specifying the latest day (in localtime) that a node should have been accessed to be kept. If not given, the current date is used.
warn: boolean, optional :
If true, echoes warning messages for all directory removed

Also used are the `PipeFunc`, callables that are returned by the *`Memory.cache()`* decorator:

class `nipype.caching.memory.`**PipeFunc**(*interface*, *base_dir*, *callback=None*)
Callable interface to nipype.interface objects
Use this to wrap nipype.interface object and call them specifying their input with keyword arguments:

```
fsl_merge = PipeFunc(fsl.Merge, base_dir='.')
out = fsl_merge(in_files=files, dimension='t')
```

Methods

__call__(**kwargs)

__init__(*interface*, *base_dir*, *callback=None*)
Parameters **interface: a nipype interface class** :
The interface class to wrap
base_dir: a string :
The directory in which the computation will be stored

callback: a callable :
An optional callable called each time after the function is called.

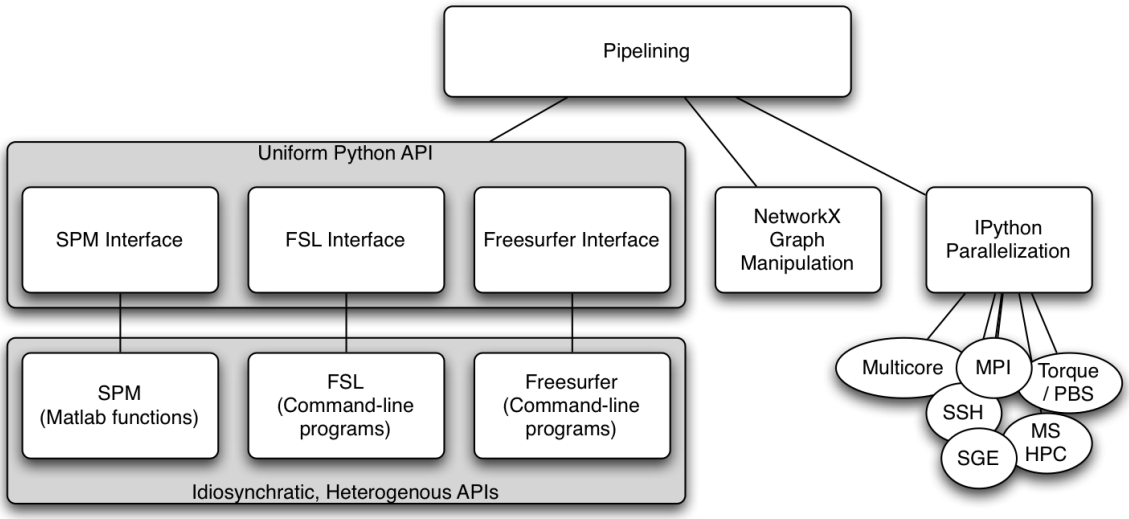
1.5 Tutorial : Workflows

This section presents several tutorials on how to setup and use pipelines. Make sure that you have the requirements satisfied and go through the steps required for the analysis tutorials.

1.5.1 Essential reading

Pipeline 101

A workflow or pipeline is built by connecting processes or nodes to each other. In the context of nipype, every interface can be treated as a pipeline node having defined inputs and outputs. Creating a workflow then is a matter of connecting appropriate outputs to inputs. Currently, workflows are limited to being directional and cannot have any loops, thereby creating an ordering to data flow. The following nipype component architecture might help understanding some of the tutorials presented here.



My first pipeline

Although the most trivial workflow consists of a single node, we will create a workflow with two nodes: a realign node that will send the realigned functional data to a smoothing node. It is important to note that setting up a workflow is separate from executing it.

1. Import appropriate modules

```
import nipype.interfaces.spm as spm          # the spm interfaces
import nipype.pipeline.engine as pe          # the workflow and node wrappers
```

2. Define nodes

Here we take instances of interfaces and make them pipeline compatible by wrapping them with pipeline specific elements. To determine the inputs and outputs of a given interface, please see *Tutorial : Interfaces*. Let's start with defining a realign node using the interface `nipype.interfaces.spm.Realign`

```
realigner = pe.Node(interface=spm.Realign(), name='realign')
realigner.inputs.in_files = 'somefuncrun.nii'
realigner.inputs.register_to_mean = True
```

This would be equivalent to:

```
realigner = pe.Node(interface=spm.Realign(infile='somefuncrun.nii',
                                          register_to_mean = True),
                    name='realign')
```

In Pythonic terms, this is saying that interface option in Node accepts an *instance* of an interface. The inputs to this interface can be set either later or while initializing the interface.

Note: In the above example, 'somefuncrun.nii' has to exist, otherwise the commands won't work. A node will check if appropriate inputs are being supplied.

Similar to the realigner node, we now set up a smoothing node.

```
smoother = pe.Node(interface=spm.Smooth(fwhm=6), name='smooth')
```

Now we have two nodes with their inputs defined. Note that we have not defined an input file for the smoothing node. This will be done by connecting the realigner to the smoother in step 5.

3. Creating and configuring a workflow

Here we create an instance of a workflow and indicate that it should operate in the current directory.

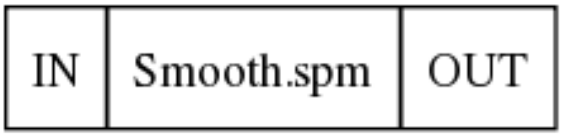
```
workflow = pe.Workflow(name='preproc')
workflow.base_dir = '.'
```

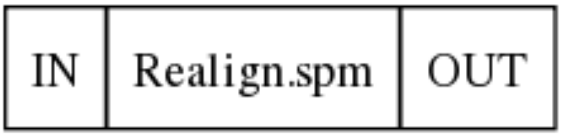
4. Adding nodes to workflows (optional)

If nodes are going to be connected (see step 5), this step is not necessary. However, if you would like to run a node by itself without connecting it to any other node, then you need to add it to the workflow. For adding nodes, order of nodes is not important.

```
workflow.add_nodes([smoother, realigner])
```

This results in a workflow containing two isolated nodes:





5. Connecting nodes to each other

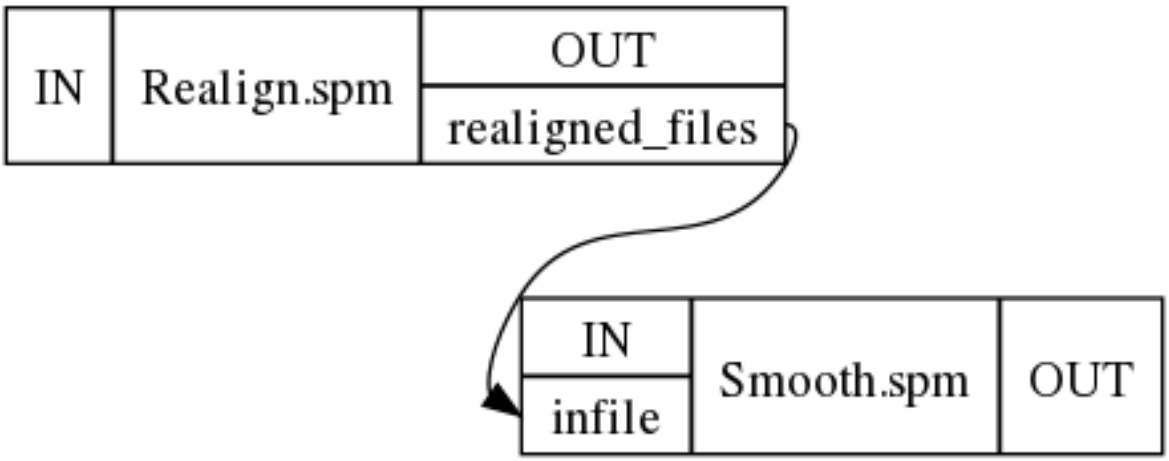
We want to connect the output produced by the node realignment to the input of the node smoothing. This is done as follows.

```
workflow.connect(realigner, 'realigned_files', smoother, 'in_files')
```

Although not shown here, the following notation can be used to connect multiple outputs from one node to multiple inputs (see step 7 below).

```
workflow.connect([(realigner, smoother, [('realigned_files', 'in_files')])])
```

This results in a workflow containing two connected nodes:



6. Visualizing the workflow

The workflow is represented as a directed acyclic graph (DAG) and one can visualize this using the following command. In fact, the pictures above were generated using this.

```
workflow.write_graph()
```

This creates two files graph.dot and graph_detailed.dot and if graphviz is installed on your system it automatically converts it to png files. If graphviz is not installed you can take the dot files and load them in a graphviz visualizer elsewhere. You can specify how detailed the graph is going to be, by using "graph2use" argument which takes the following options:

- hierarchical - creates a graph showing all embedded workflows (default)
- orig - creates a top level graph without expanding internal workflow nodes
- flat - expands workflow nodes recursively
- exec - expands workflows to depict iterables (be careful - can generate really large graphs)

7. Extend it

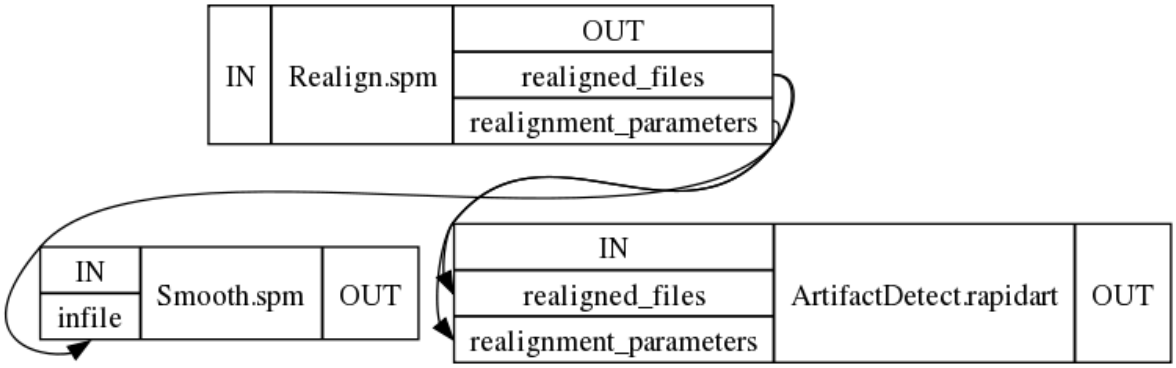
Now that you have seen a basic pipeline let's add another node to the above pipeline.

```
import nipype.algorithms.rapidart as ra
artdetect = pe.Node(interface=ra.ArtifactDetect(), name='artdetect')
artdetect.inputs.use_differences  = [True, False]
art.inputs.use_norm = True
art.inputs.norm_threshold = 0.5
art.inputs.zintensity_threshold = 3
workflow.connect([(realigner, artdetect,
                   [('realigned_files', 'realigned_files'),
                    ('realignment_parameters','realignment_parameters')]
                  )])
```

Note: a) How an alternative form of connect was used to connect multiple output fields from the realign node to corresponding input fields of the artifact detection node.

b) The current visualization only shows connected input and output ports. It does not show all the parameters that you have set for a node.

This results in



8. Execute the workflow
Assuming that **somefuncrun.nii** is actually a file or you've replaced it with an appropriate one, you can run the pipeline with:

```
workflow.run()
```

This should create a folder called preproc in your current directory, inside which are three folders: realign, smooth and artdetect (the names of the nodes). The outputs of these routines are in these folders.
pipeline Connected series of processes (processes can be run parallel and or sequential)
workflow (kind of synonymous to pipeline) = hosting the nodes
node = switching-point within a pipeline, you can give it a name (in the above example e.g. realigner), a node usually requires an or several inputs and will produce an or several outputs
interface = specific software (e.g. FSL, SPM ...) are wrapped in interfaces, within a node instances of an interface can be run
modules for each interface the according modules have to be imported in the usual pythonic manner

Pipeline 102

Now that you know how to construct a workflow and execute it, we will go into more advanced concepts. This tutorial focuses on `nipype.pipeline.engine.Workflow` `nipype.pipeline.engine.Node` and `nipype.pipeline.engine.MapNode`.
A workflow is a **directed acyclic graph (DAG)** consisting of nodes which can be of type *Workflow*, *Node* or *MapNode*. Workflows can be re-used and hierarchical workflows can be easily constructed.

'name' : the mandatory keyword arg

When instantiating a Workflow, Node or MapNode, a *name* has to be provided. For any given level of a workflow, no two nodes can have the same name. The engine will let you know if this is the case when you add nodes to a workflow either directly using *add_nodes* or using the *connect* function.
Names have many internal uses. They determine the name of the directory in which the workflow/node is run and the outputs are stored.

```
realigner = pe.Node(interface=spm.Realign(),
                    name='RealignSPM')
```

Now this output will be stored in a directory called *RealignSPM*. Proper naming of your nodes can be advantageous from the perspective that it provides a semantic descriptor aligned with your thought process. This name parameter is also used to refer to nodes in embedded workflows.

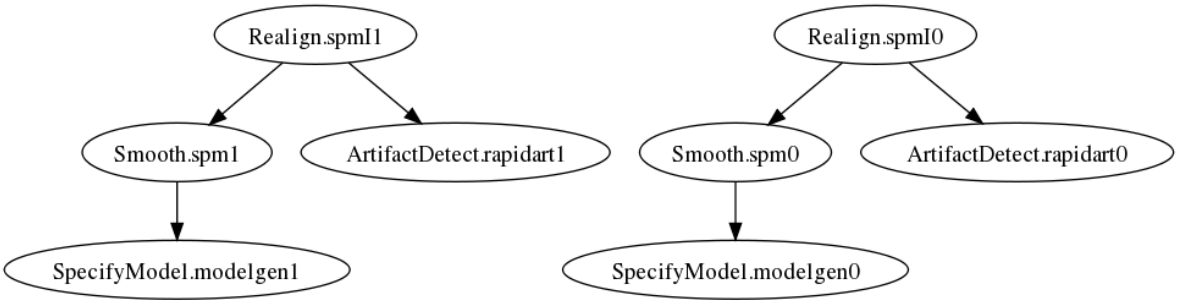
iterables This can only be set for Node and MapNode. This is syntactic sugar for running a subgraph with the Node/MapNode at its root in a `for` loop. For example, consider an fMRI preprocessing pipeline that you would like to run for all your subjects. You can define a workflow and then execute it for every single subject inside a `for` loop. Consider the simplistic example below, where startnode is a node belonging to workflow 'mywork.'

```
for s in subjects:
    startnode.inputs.subject_id = s
    mywork.run()
```

The pipeline engine provides a convenience function that simplifies this:

```
startnode.iterables = ('subject_id', subjects)
mywork.run()
```

This will achieve the same exact behavior as the for loop above. The workflow graph is:



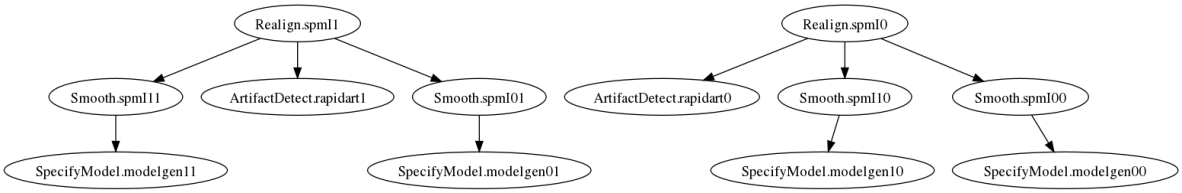
Now consider the situation in which you want the last node (typically smoothing) of your preprocessing pipeline to smooth using two different kernels (0 mm and 6 mm FWHM). Again the common approach would be:

```
for s in subjects:
    startnode.inputs.subject_id = s
    uptosmoothingworkflow.run()
    smoothnode.inputs.infile = lastnode.output.outfile
    for fwhm in [0, 6]:
        smoothnode.inputs.fwhm = fwhm
        remainingworkflow.run()
```

Instead of having multiple `for` loops at various stages, you can set up another set of iterables for the smoothnode.

```
startnode.iterables = ('subject_id', subjects)
smoothnode.iterables = ('fwhm', [0, 6])
mywork.run()
```

This will run the preprocessing workflow for two different smoothing kernels over all subjects.



Thus setting iterables has a multiplicative effect. In the above examples there is a separate, distinct specifymodel node that's executed for each combination of subject and smoothing.

iterfield This is a mandatory keyword arg for MapNode. This enables running the underlying interface over a set of inputs and is particularly useful when the interface can only operate on a single input. For example, the `nipype.interfaces.fsl.BET` will operate on only one (3d or 4d) NIfTI file. But wrapping BET in a MapNode can execute it over a list of files:

```
better = pe.MapNode(interface=fsl.Bet(), name='stripper',
                     iterfield=['in_file'])
better.inputs.in_file = ['file1.nii','file2.nii']
better.run()
```

This will create a directory called `stripper` and inside it two subdirectories called `in_file_0` and `in_file_1`. The output of running bet separately on each of those files will be stored in those two subdi-

rectories.
This can be extended to run it on pairwise inputs. For example,

```
transform = pe.MapNode(interface=fs.ApplyVolTransform(),
                       name='warpvol',
                       iterfield=['source_file', 'reg_file'])
transform.inputs.source_file = ['file1.nii','file2.nii']
transform.inputs.reg_file = ['file1.reg','file2.reg']
transform.run()
```

The above will be equivalent to running transform by taking corresponding items from each of the two fields in iterfield. The subdirectories get always named with respect to the first iterfield.

overwrite The overwrite keyword arg forces a node to be rerun.

The *clone* function The *clone* function can be used to create a copy of a workflow. No references to the original workflow are retained. As such the clone function requires a name keyword arg that specifies a new name for the duplicate workflow.

Pipeline 103

Modifying inputs to pipeline nodes

Two nodes can be connected as shown below.

```
workflow.connect(realigner, 'realigned_files', smoother, 'infile')
```

The connection mechanism allows for a function to be evaluated on the output field ('realigned files') of the source node (realigner) and have its result be sent to the input field ('infile') of the destination node (smoother).

```
def reverse_order(inlist):
    inlist.reverse()
    return inlist

workflow.connect(realigner, ('realigned_files', reverse_order),
                 smoother, 'infile')
```

This can be extended to provide additional arguments to the function. For example:

```
def reorder(inlist, order):
   return [inlist[item] for item in order]

workflow.connect(realigner, ('realigned_files', reorder, [2, 3, 0, 1]),
                 smoother, 'infile')
```

In this example, we assume the realigned_files produces a list of 4 files. We can reorder these files in a particular order using the modifier. Since such modifications are not tracked, they should be used with extreme care and only in cases where absolutely necessary. Often, one may find that it is better to insert a node rather than a function.

Distributed computation

The pipeline engine has built-in support for distributed computation on clusters. This can be achieved via plugin-modules for Python multiprocessing or the IPython distributed computing interface or SGE/PBS/Condor, provided the user sets up a workflow on a shared filesystem. These modules can take arguments that specify additional distribution engine parameters. For IPython the environment needs to be configured for distributed operation. Details are available at *Using Nipype Plugins*.
The default behavior is to run in series using the Linear plugin.

```
workflow.run()
```

In some cases it may be advantageous to run the workflow in series locally (e.g., debugging, small-short pipelines, large memory only interfaces, relocating working directory/updating hashes).

Debugging

When a crash happens while running a pipeline, a crashdump is stored in the pipeline's working directory unless the config option 'crashdumpdir' has been set (see :ref:config_options).
The crashdump is a compressed numpy file that stores a dictionary containing three fields:

1. node - the node that failed
2. execgraph - the graph that the node came from
3. traceback - from local or remote session for the failure.

We keep extending the information contained in the file and making it easier to troubleshoot the failures. However, in the meantime the following can help to recover information related to the failure.
in IPython do (`%pdb` in IPython is similar to `dbstop` if error in Matlab):

```
from nipype.utils.filemanip import loadflat
crashinfo = loadflat('crashdump....npz')
%pdb
crashinfo['node'].run()  # re-creates the crash
pdb> up  #typically, but not necessarily the crash is one stack frame up
pdb> inspect variables
pdb>quit
```

Relocation of workdir

In some circumstances, one might decide to move their entire working directory to a new location. It would be convenient to rerun only necessary components of the pipeline, instead of running all the nodes all over again. It is possible to do that with the `updatehash()` function.

```
workflow.run(updatehash=True)
```

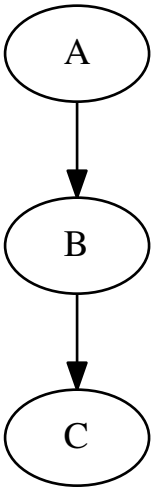
This will execute the workflow and update all the hash values that were stored without actually running any of the interfaces.

MapNode, iterfield, and iterables explained

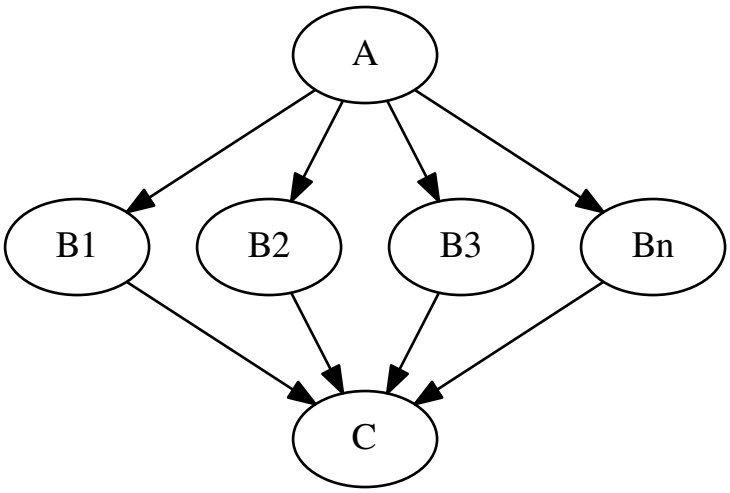
In this chapter we will try to explain the concepts behind MapNode, iterfield, and iterables.

MapNode and iterfield

Imagine that you have a list of items (lets say files) and you want to execute the same node on them (for example some smoothing or masking). Some nodes accept multiple files and do exactly the same thing on them, but some don't (they expect only one file). MapNode can solve this problem. Imagine you have the following workflow:



Node "A" outputs a list of files, but node "B" accepts only one file. Additionally "C" expects a list of files. What you would like is to run "B" for every file in the output of "A" and collect the results as a list and feed it to "C". Something like this:



The code to achieve this is quite simple

```
import nipype.pipeline.engine as pe
a = pe.Node(interface=A(), name="a")
b = pe.MapNode(interface=B(), name="b", iterfield=['in_file'])
c = pe.Node(interface=C(), name="c")

my_workflow = pe.Workflow(name="my_workflow")
```

```
my_workflow.connect([(a,b,[('out_files','in_file')]),
                     (b,c,[('out_file','in_files')])
                     ])
```

assuming that interfaces "A" and "C" have one input "in_files" and one output "out_files" (both lists of files). Interface "B" has single file input "in_file" and single file output "out_file".

You probably noticed that you connect nodes as if "B" could accept and output list of files. This is because it is wrapped using MapNode instead of Node. This special version of node will (under the bonnet) create an instance of "B" for every item in the list from the input. The compulsory argument "iterfield" defines which input should it iterate over (for example in single file smooth interface you would like to iterate over input files not the smoothing width). At the end outputs are collected into a list again. In other words this is map and reduce scenario.

You might have also noticed that the iterfield arguments expects a list of input names instead of just one name. This suggests that there can be more than one! Even though a bit confusing this is true. You can specify more than one input to iterate over but the lists that you provide (for all the inputs specified in iterfield) have to have the same length. MapNode will then pair the parameters up and run the first instance with first set of parameters and second with second set of parameters. For example, this code:

```
b = pe.MapNode(interface=B(), name="b", iterfield=['in_file', 'n'])
b.inputs.in_file = ['file', 'another_file', 'different_file']
b.inputs.n = [1,2,3]
b.run()
```

is almost the same as running

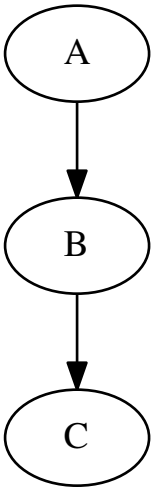
```
b1 = pe.Node(interface=B(), name="b1")
b1.inputs.in_file = 'file'
b1.inputs.n = 1

b2 = pe.Node(interface=B(), name="b2")
b2.inputs.in_file = 'another_file'
b2.inputs.n = 2

b3 = pe.Node(interface=B(), name="b3")
b3.inputs.in_file = 'different_file'
b3.inputs.n = 3
```

It is a rarely used feature, but you can sometimes find it useful.

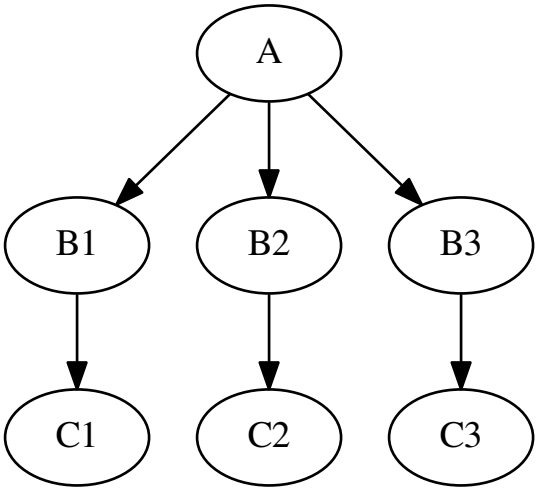
In more advanced applications it is useful to be able to iterate over items of nested lists (for example [[1,2],[3,4]]). MapNode allows you to do this with the "nested=True" parameter. Outputs will preserve the same nested structure as the inputs.

Iterables

Now imagine a different scenario. You have your workflow as before



and there are three possible values of one of the inputs node "B" you would like to investigate (for example width of 2,4, and 6 pixels of a smoothing node). You would like to see how different parameters in node "B" would influence everything that depends on its outputs (node "C" in our example). Therefore the new graph should look like this:



Of course you can do it manually by creating copies of all the nodes for different parameter set, but this can be very time consuming, especially when there are more than one node taking inputs from "B". Luckily nipype supports this scenario! Its called iterables and and you use it this way:

```
import nipype.pipeline.engine as pe
a = pe.Node(interface=A(), name="a")
b = pe.Node(interface=B(), name="b")
```

```
b.iterables = ("n", [1, 2, 3])
c = pe.Node(interface=C(), name="c")

my_workflow = pe.Workflow(name="my_workflow")
my_workflow.connect([(a,b,[('out_file','in_file')]),
                     (b,c,[('out_file','in_file')])
                     ])
```

Assuming that you want to try out values 1, 2, and 3 of input "n" of the node "B". This will also create three different versions of node "C" - each with inputs from instances of node "C" with different values of "n". Additionally, you can set multiple iterables for a node with a list of tuples in the above format.
Iterables are commonly used to execute the same workflow for many subjects. Usually one parametrises DataGrabber node with subject ID. This is achieved by connecting an IdentityInterface in front of DataGrabber. When you set iterables of the IdentityInterface to the list of subjects IDs, the same workflow will be executed for every subject. See `examples/fmri_spm` to see this pattern in action.

DataGrabber and DataSink explained

In this chapter we will try to explain the concepts behind DataGrabber and DataSink.

Why do we need these interfaces?

A typical workflow takes data as input and produces data as the result of one or more operations. One can set the data required by a workflow directly as illustrated below.

```
from fsl_tutorial2 import preproc
preproc.base_dir = os.path.abspath('.')
preproc.inputs.inputspec.func = os.path.abspath('data/s1/f3.nii')
preproc.inputs.inputspec.struct = os.path.abspath('data/s1/struct.nii')
preproc.run()
```

Typical neuroimaging studies require running workflows on multiple subjects or different parameterizations of algorithms. One simple approach to that would be to simply iterate over subjects.

```
from fsl_tutorial2 import preproc
for name in subjects:
  preproc.base_dir = os.path.abspath('.')
  preproc.inputs.inputspec.func = os.path.abspath('data/%s/f3.nii'%name)
  preproc.inputs.inputspec.struct = os.path.abspath('data/%s/struct.nii'%name)
  preproc.run()
```

However, in the context of complex workflows and given that users typically arrange their imaging and other data in a semantically hierarchical data store, an alternative mechanism for reading and writing the data generated by a workflow is often necessary. As the names suggest DataGrabber is used to get at data stored in a shared file system while DataSink is used to store the data generated by a workflow into a hierarchical structure on disk.

DataGrabber

DataGrabber is an interface for collecting files from hard drive. It is very flexible and supports almost any file organization of your data you can imagine.
You can use it as a trivial use case of getting a fixed file. By default, DataGrabber stores its outputs in a field called outfiles.

```
import nipype.interfaces.io as nio
datasource1 = nio.DataGrabber()
datasource1.inputs.base_directory = os.getcwd()
datasource1.inputs.template = 'data/s1/f3.nii'
results = datasource1.run()
```

Or you can get at all uncompressed NIfTI files starting with the letter 'f' in all directories starting with the letter 's'.

```
datasource2.inputs.base_directory = '/mass'
datasource2.inputs.template = 'data/s*/f*.nii'
```

Two special inputs were used in these previous cases. The input *base_directory* indicates in which directory to search, while the input *template* indicates the string template to match. So in the previous case datagrabber is looking for path matches of the form */mass/data/s*/f**.

Note: When used with wildcards (e.g., s* and f* above) DataGrabber does not return data in sorted order. In order to force it to return data in sorted order, one needs to set the input *sorted* = *True*. However, when explicitly specifying an order as we will see below, *sorted* should be set to *False*.

More useful cases arise when the template can be filled by other inputs. In the example below, we define an input field for *datagrabber* called *run*. This is then used to set the template (see %d in the template).

```
datasource3 = nio.DataGrabber(infields=['run'])
datasource3.inputs.base_directory = os.getcwd()
datasource3.inputs.template = 'data/s1/f%d.nii'
datasource3.inputs.run = [3, 7]
```

This will return files *basedir/data/s1/f3.nii* and *basedir/data/s1/f7.nii*. We can take this a step further and pair subjects with runs.

```
datasource4 = nio.DataGrabber(infields=['subject_id', 'run'])
datasource4.inputs.template = 'data/%s/f%d.nii'
datasource4.inputs.run = [3, 7]
datasource4.inputs.subject_id = ['s1', 's3']
```

This will return files *basedir/data/s1/f3.nii* and *basedir/data/s3/f7.nii*.

A more realistic use-case In a typical study one often wants to grab different files for a given subject and store them in semantically meaningful outputs. In the following example, we wish to retrieve all the functional runs and the structural image for the subject 's1'.

```
datasource = nio.DataGrabber(infields=['subject_id'], outfields=['func', 'struct'])
datasource.inputs.base_directory = 'data'
datasource.inputs.template = '*'
datasource.inputs.field_template = dict(func='%s/f%d.nii',
                                        struct='%s/struct.nii')
datasource.inputs.template_args = dict(func=[['subject_id', [3,5,7,10]]],
                                       struct=[['subject_id']])
datasource.inputs.subject_id = 's1'
```

Two more fields are introduced: *field_template* and *template_args*. These fields are both dictionaries whose keys correspond to the *outfields* keyword. The *field_template* reflects the search path for each output field, while the *template_args* reflect the inputs that satisfy the template. The inputs can either be one of the named inputs specified by the *infields* keyword arg or it can be raw strings or integers corresponding to the template. For the *func* output, the **%s** in the *field_template* is satisfied by *subject_id* and the **%d** is field in by the list of numbers.

Note: We have not set *sorted* to *True* as we want the DataGrabber to return the functional files in the order it was specified rather than in an alphabetic sorted order.

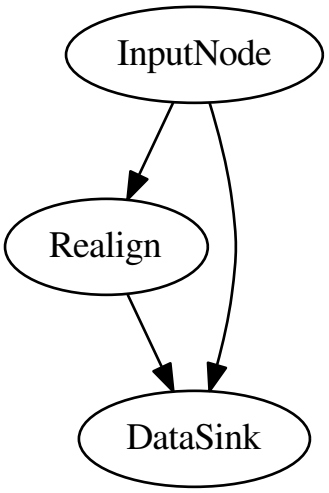
DataSink

A workflow working directory is like a **cache**. It contains not only the outputs of various processing stages, it also contains various extraneous information such as execution reports, hashfiles determining the input state of processes. All of this is embedded in a hierarchical structure that reflects the iterables that have been used in the

workflow. This makes navigating the working directory a not so pleasant experience. And typically the user is interested in preserving only a small percentage of these outputs. The DataSink interface can be used to extract components from this *cache* and store it at a different location. For XNAT-based storage, see XNATSink .

Note: Unlike other interfaces, a DataSink's inputs are defined and created by using the workflow connect statement. Currently disconnecting an input from the DataSink does not remove that connection port.

Let's assume we have the following workflow.



The following code segment defines the DataSink node and sets the *base_directory* in which all outputs will be stored. The *container* input creates a subdirectory within the *base_directory*. If you are iterating a workflow over subjects, it may be useful to save it within a folder with the subject id.

```
datasink = pe.Node(nio.DataSink(), name='sinker')
datasink.inputs.base_directory = '/path/to/output'
workflow.connect(inputnode, 'subject_id', datasink, 'container')
```

If we wanted to save the realigned files and the realignment parameters to the same place the most intuitive option would be:

```
workflow.connect(realigner, 'realigned_files', datasink, 'motion')
workflow.connect(realigner, 'realignment_parameters', datasink, 'motion')
```

However, this will not work as only one connection is allowed per input port. So we need to create a second port. We can store the files in a separate folder.

```
workflow.connect(realigner, 'realigned_files', datasink, 'motion')
workflow.connect(realigner, 'realignment_parameters', datasink, 'motion.par')
```

The period (.) indicates that a subfolder called par should be created. But if we wanted to store it in the same folder as the realigned files, we would use the .@ syntax. The @ tells the DataSink interface to not create the subfolder. This will allow us to create different named input ports for DataSink and allow the user to store the files in the same folder.

```
workflow.connect(realigner, 'realigned_files', datasink, 'motion')
workflow.connect(realigner, 'realignment_parameters', datasink, 'motion.@par')
```

The syntax for the input port of DataSink takes the following form:

```
string[[.[@]]string[[.[@]]string] ...]
where parts between paired [] are optional.
```

MapNode In order to use DataSink inside a MapNode, it's inputs have to be defined inside the constructor using the *infields* keyword arg.

Parameterization As discussed in MapNode, iterfield, and iterables explained, one can run a workflow iterating over various inputs using the iterables attribute of nodes. This means that a given workflow can have multiple outputs depending on how many iterables are there. Iterables create working directory subfolders such as *_iterable_name_value*. The *parameterization* input parameter controls whether the data stored using DataSink is in a folder structure that contains this iterable information or not. It is generally recommended to set this to *True* when using multiple nested iterables.

Substitutions The *substitutions* and *substitutions_regexp* inputs allow users to modify the output destination path and name of a file. Substitutions are a list of 2-tuples and are carried out in the order in which they were entered. Assuming that the output path of a file is:

```
/root/container/_variable_1/file_subject_realigned.nii
```

we can use substitutions to clean up the output path.

```
datasink.inputs.substitutions = [('_variable', 'variable'),
                                 ('file_subject_', '')]
```

This will rewrite the file as:

```
/root/container/variable_1/realigned.nii
```

Note: In order to figure out which substitutions are needed it is often useful to run the workflow on a limited set of iterables and then determine the substitutions.

1.5.2 Beginner's guide

By Michael Notter. Available here

1.5.3 Example workflows

1.5.4 Requirements

All tutorials
Release 0.4 of nipype and it's dependencies have been installed
Analysis tutorials
FSL, FreeSurfer, Camino, ConnectomeViewer and MATLAB are available and callable from the command line
SPM 5/8 is installed and callable in matlab
Space: 3-10 GB

1.5.5 Checklist for analysis tutorials

For the analysis tutorials, we will be using a slightly modified version of the FBIRN Phase I travelling data set.

Step 0

Download and extract the Pipeline tutorial data (429MB).
(checksum: 56ed4b7e0aac5627d1724e9c10cd26a7)

Step 1.

Ensure that all programs are available by calling `bet`, `matlab` and then `which spm` within matlab to ensure you have spm5/8 in your matlab path.

Step 2.

You can now run the tutorial by typing `python tutorial_script.py` within the nipype-tutorial directory. This will run a full first level analysis on two subjects following by a 1-sample t-test on their first level results. The next section goes through each section of the tutorial script and describes what it is doing.

1.6 Using Nipype Plugins

The workflow engine supports a plugin architecture for workflow execution. The available plugins allow local and distributed execution of workflows and debugging. Each available plugin is described below.
Current plugins are available for Linear, Multiprocessing, IPython distributed processing platforms and for direct processing on SGE, PBS, HTCondor, LSF, and SLURM. We anticipate future plugins for the Soma workflow.

Note: The current distributed processing plugins rely on the availability of a shared filesystem across computational nodes.
A variety of config options can control how execution behaves in this distributed context. These are listed later on in this page.

All plugins can be executed with:

```
workflow.run(plugin=PLUGIN_NAME, plugin_args=ARGS_DICT)
```

Optional arguments:

```
status_callback : a function handle
max_jobs : maximum number of concurrent jobs
max_tries : number of times to try submitting a job
retry_timeout : amount of time to wait between tries
```

Note: Except for the status_callback, the remaining arguments only apply to the distributed plugins: MultiProc/IPython(X)/SGE/PBS/HTCondor/HTCondorDAGMan/LSF

For example:

1.6.1 Plugins

Debug

This plugin provides a simple mechanism to debug certain components of a workflow without executing any node.
Mandatory arguments:

```
callable :  A function handle that receives as arguments a node and a graph
```

The function callable will called for every node from a topological sort of the execution graph.

Linear

This plugin runs the workflow one node at a time in a single process locally. The order of the nodes is determined by a topological sort of the workflow:

```
workflow.run(plugin='Linear')
```

MultiProc

Uses the Python multiprocessing library to distribute jobs as new processes on a local system.
Optional arguments:

```
n_procs :  Number of processes to launch in parallel, if not set number of
processors/threads will be automatically detected
```

To distribute processing on a multicore machine, simply call:

```
workflow.run(plugin='MultiProc')
```

This will use all available CPUs. If on the other hand you would like to restrict the number of used resources (to say 2 CPUs), you can call:

```
workflow.run(plugin='MultiProc', plugin_args={'n_procs' : 2}
```

IPython

This plugin provide access to distributed computing using IPython parallel machinery.

Note: We provide backward compatibility with IPython versions earlier than 0.10.1 using the IPythonX plugin. Please read the IPython documentation to determine how to setup your cluster for distributed processing. This typically involves calling ipcluster.

Once the clients have been started, any pipeline executed with:

```
workflow.run(plugin='IPython')
```

SGE/PBS

In order to use nipype with SGE or PBS you simply need to call:

```
workflow.run(plugin='SGE')
workflow.run(plugin='PBS')
```

Optional arguments:

```
template: custom template file to use
qsub_args: any other command line args to be passed to qsub.
max_jobname_len: (PBS only) maximum length of the job name.  Default 15.
```

For example, the following snippet executes the workflow on myqueue with a custom template:

```
workflow.run(plugin='SGE',
   plugin_args=dict(template='mytemplate.sh', qsub_args='-q myqueue')
```

In addition to overall workflow configuration, you can use node level configuration for PBS/SGE:

```
node.plugin_args = {'qsub_args': '-l nodes=1:ppn=3'}
```

this would apply only to the node and is useful in situations, where a particular node might use more resources than other nodes in a workflow.

Note: Setting the keyword *overwrite* would overwrite any global configuration with this local configuration:

```
node.plugin_args = {'qsub_args': '-l nodes=1:ppn=3', 'overwrite': True}
```

SGEGraph

SGEGraph is an execution plugin working with Sun Grid Engine that allows for submitting entire graph of dependent jobs at once. This way Nipype does not need to run a monitoring process - SGE takes care of this. The use of *SGEGraph* is preferred over SGE since the latter adds unnecessary load on the submit machine.

Note: When rerunning unfinished workflows using SGEGraph you may decide not to submit jobs for Nodes that previously finished running. This can speed up execution, but new or modified inputs that would previously trigger a Node to rerun will be ignored. The following option turns on this functionality:

```
workflow.run(plugin='SGEGraph', plugin_args = {'dont_resubmit_completed_jobs': True})
```

LSF

Submitting via LSF is almost identical to SGE above above except for the optional arguments field:

```
workflow.run(plugin='LSF')
```

Optional arguments:

```
template: custom template file to use
bsub_args: any other command line args to be passed to bsub.
```

SLURM

Submitting via SLURM is almost identical to SGE above except for the optional arguments field:
workflow.run(plugin='SLURM')
Optional arguments:

```
template: custom template file to use
sbatch_args: any other command line args to be passed to bsub.
```

SLURMGraph

SLURMGraph is an execution plugin working with SLURM that allows for submitting entire graph of dependent jobs at once. This way Nipype does not need to run a monitoring process - SLURM takes care of this. The use of *SLURMGraph* plugin is preferred over the vanilla SLURM plugin since the latter adds unnecessary load on the submit machine.

Note: When rerunning unfinished workflows using SLURMGraph you may decide not to submit jobs for Nodes that previously finished running. This can speed up execution, but new or modified inputs that would previously trigger a Node to rerun will be ignored. The following option turns on this functionality:

```
workflow.run(plugin='SLURMGraph', plugin_args = {'dont_resubmit_completed_jobs': True})
```

HTCondor

DAGMan

With its DAGMan component HTCondor (previously Condor) allows for submitting entire graphs of dependent jobs at once (similar to *SGEGraph* and *SLURMGraph*). With the `CondorDAGMan` plug-in Nipype can utilize this functionality to submit complete workflows directly and in a single step. Consequently, and in contrast to other plug-ins, workflow execution returns almost instantaneously – Nipype is only used to generate the workflow graph, while job scheduling and dependency resolution are entirely managed by HTCondor.
Please note that although DAGMan supports specification of data dependencies as well as data provisioning on compute nodes this functionality is currently not supported by this plug-in. As with all other batch systems supported by Nipype, only HTCondor pools with a shared file system can be used to process Nipype workflows.
Workflow execution with HTCondor DAGMan is done by calling:

```
workflow.run(plugin='CondorDAGMan')
```

Job execution behavior can be tweaked with the following optional plug-in arguments. The value of most arguments can be a literal string or a filename, where in the latter case the content of the file will be used as the argument value:

```
submit_template : submit spec template for individual jobs in a DAG (see
            CondorDAGManPlugin.default_submit_template for the default.
initial_specs : additional submit specs that are prepended to any job's
            submit file
override_specs : additional submit specs that are appended to any job's
            submit file
wrapper_cmd : path to an exectuable that will be started instead of a node
            script. This is useful for wrapper script that execute certain
            functionality prior or after a node runs. If this option is
            given the wrapper command is called with the respective Python
            exectuable and the path to the node script as final arguments
wrapper_args : optional additional arguments to a wrapper command
dagman_args : arguments to be prepended to the job execution script in the
             dagman call
block : if True the plugin call will block until Condor has finished
        prcoessing the entire workflow (default: False)
```

Please see the HTCondor documentation for details on possible configuration options and command line arguments.

Using the `wrapper_cmd` argument it is possible to combine Nipype workflow execution with checkpoint/migration functionality offered by, for example, DMTCP. This is especially useful in the case of workflows with long running nodes, such as Freesurfer's recon-all pipeline, where Condor's job prioritization algorithm could lead to jobs being evicted from compute nodes in order to maximize overall troughput. With checkpoint/migration enabled such a job would be checkpointed prior eviction and resume work from the checkpointed state after being rescheduled – instead of restarting from scratch.

On a Debian system, executing a workflow with support for checkpoint/migration for all nodes could look like this:

```
# define common parameters
dmtcp_hdr = """
should_transfer_files = YES
when_to_transfer_output = ON_EXIT_OR_EVICT
kill_sig = 2
environment = DMTCP_TMPDIR=./;JALIB_STDERR_PATH=/dev/null;DMTCP_PREFIX_ID=$(CLUSTER)_$(PROCESS)
"""
shim_args = "--log %(basename)s.shimlog --stdout %(basename)s.shimout --stderr %(basename)s.shim
# run workflow
workflow.run(
      plugin='CondorDAGMan',
      plugin_args=dict(initial_specs=dmtcp_hdr,
                       wrapper_cmd='/usr/lib/condor/shim_dmtcp',
                       wrapper_args=shim_args)
      )
```

`qsub` emulation

Note: This plug-in is deprecated and users should migrate to the more robust and more versatile `CondorDAGMan` plug-in.

Despite the differences between HTCondor and SGE-like batch systems the plugin usage (incl. supported arguments) is almost identical. The HTCondor plugin relies on a `qsub` emulation script for HTCondor, called `condor_qsub` that can be obtained from a Git repository on git.debian.org. This script is currently not shipped with a standard HTCondor distribution, but is included in the HTCondor package from http://neuro.debian.net.

It is sufficient to download this script and install it in any location on a system that is included in the `PATH` configuration.
Running a workflow in a HTCondor pool is done by calling:

```
workflow.run(plugin='Condor')
```

The plugin supports a limited set of qsub arguments (`qsub_args`) that cover the most common use cases. The `condor_qsub` emulation script translates qsub arguments into the corresponding HTCondor terminology and handles the actual job submission. For details on supported options see the manpage of `condor_qsub`.
Optional arguments:

```
qsub_args: any other command line args to be passed to condor_qsub.
```

1.7 Configuration File

Some of the system wide options of Nipype can be configured using a configuration file. Nipype looks for the file in the local folder under the name `nipype.cfg` and in `~/.nipype/nipype.cfg` (in this order). If an option will not be specified a default value will be assumed. The file is divided into following sections:

1.7.1 Logging

workflow_level How detailed the logs regarding workflow should be (possible values: `INFO` and `DEBUG`; default value: `INFO`)

filemanip_level How detailed the logs regarding file operations (for example overwriting warning) should be (possible values: `INFO` and `DEBUG`; default value: `INFO`)

interface_level How detailed the logs regarding interface execution should be (possible values: `INFO` and `DEBUG`; default value: `INFO`)

log_to_file Indicates whether logging should also send the output to a file (possible values: `true` and `false`; default value: `false`)

log_directory Where to store logs. (string, default value: home directory)

log_size Size of a single log file. (integer, default value: 254000)

log_rotate How many rotation should the log file make. (integer, default value: 4)

1.7.2 Execution

plugin This defines which execution plugin to use. (possible values: `Linear`, `MultiProc`, `SGE`, `IPython`; default value: `Linear`)

stop_on_first_crash Should the workflow stop upon first node crashing or try to execute as many nodes as possible? (possible values: `true` and `false`; default value: `false`)

stop_on_first_rerun Should the workflow stop upon first node trying to recompute (by that we mean rerunning a node that has been run before - this can happen due changed inputs and/or hash_method since the last run). (possible values: `true` and `false`; default value: `false`)

hash_method Should the input files be checked for changes using their content (slow, but 100% accurate) or just their size and modification date (fast, but potentially prone to errors)? (possible values: `content` and `timestamp`; default value: `content`)

keep_inputs Ensures that all inputs that are created in the nodes working directory are kept after node execution (possible values: `true` and `false`; default value: `false`)

single_thread_matlab Should all of the Matlab interfaces (including SPM) use only one thread? This is useful if you are parallelizing your workflow using MultiProc or IPython on a single multicore machine. (possible values: `true` and `false`; default value: `true`)

display_variable What `DISPLAY` variable should all command line interfaces be run with. This is useful if you are using xnest or Xvfb and you would like to redirect all spawned windows to it. (possible values: any X server address; default value: not set)

remove_unnecessary_outputs This will remove any interface outputs not needed by the workflow. If the required outputs from a node changes, rerunning the workflow will rerun the node. Outputs of leaf nodes

(nodes whose outputs are not connected to any other nodes) will never be deleted independent of this parameter. (possible values: `true` and `false`; default value: `true`)

try_hard_link_datasink When the DataSink is used to produce an orginized output file outside of nipypes internal cache structure, a file system hard link will be attempted first. A hard link allow multiple file paths to point to the same physical storage location on disk if the condisions allow. By refering to the same physical file on disk (instead of copying files byte-by-byte) we can avoid unnecessary data duplication. If hard links are not supported for the source or destination paths specified, then a standard byte-by-byte copy is used. (possible values: `true` and `false`; default value: `true`)

use_relative_paths Should the paths stored in results (and used to look for inputs) be relative or absolute. Relative paths allow moving the whole working directory around but may cause problems with symlinks. (possible values: `true` and `false`; default value: `false`)

local_hash_check Perform the hash check on the job submission machine. This option minimizes the number of jobs submitted to a cluster engine or a multiprocessing pool to only those that need to be rerun. (possible values: `true` and `false`; default value: `true`)

job_finished_timeout When batch jobs are submitted through, SGE/PBS/Condor they could be killed externally. Nipype checks to see if a results file exists to determine if the node has completed. This timeout determines for how long this check is done after a job finish is detected. (float in seconds; default value: 5)

remove_node_directories (EXPERIMENTAL) Removes directories whose outputs have already been used up. Doesn't work with IdentiInterface or any node that patches data through (without copying) (possible values: `true` and `false`; default value: `false`)

stop_on_unknown_version If this is set to True, an underlying interface will raise an error, when no version information is available. Please notify developers or submit a patch.

parameterize_dirs If this is set to True, the node's output directory will contain full parameterization of any iterable, otherwise parameterizations over 32 characters will be replaced by their hash. (possible values: `true` and `false`; default value: `true`)

poll_sleep_duration This controls how long the job submission loop will sleep between submitting all pending jobs and checking for job completion. To be nice to cluster schedulers the default is set to 60 seconds.

xvfb_max_wait Maximum time (in seconds) to wait for Xvfb to start, if the _redirect_x parameter of an Interface is True.

1.7.3 Example

```
[logging]
workflow_level = DEBUG

[execution]
stop_on_first_crash = true
hash_method = timestamp
display_variable = :1
```

Workflow.config property has a form of a nested dictionary reflecting the structure of the .cfg file.

```
myworkflow = pe.Workflow()
myworkflow.config['execution'] = {'stop_on_first_rerun': 'True',
                                  'hash_method': 'timestamp'}
```

You can also directly set global config options in your workflow script. An example is shown below. This needs to be called before you import the pipeline or the logger. Otherwise logging level will not be reset.

```
from nipype import config
cfg = dict(logging=dict(workflow_level = 'DEBUG'),
           execution={'stop_on_first_crash': False,
                      'hash_method': 'content'})
config.update_config(cfg)
```

1.7.4 Enabling logging to file

By default, logging to file is disabled. One can enable and write the file to a location of choice as in the example below.

```
import os
from nipype import config, logging
config.update_config({'logging': {'log_directory': os.getcwd(),
                                  'log_to_file': True}})
logging.update_logging(config)
```

The logging update line is necessary to change the behavior of logging such as output directory, logging level, etc.,.

1.7.5 Debug configuration

To enable debug mode, one can insert the following lines:

```
from nipype import config, logging
config.enable_debug_mode()
logging.update_logging(config)
```

In this mode the following variables are set:

```
config.set('execution', 'stop_on_first_crash', 'true')
config.set('execution', 'remove_unnecessary_outputs', 'false')
config.set('logging', 'workflow_level', 'DEBUG')
config.set('logging', 'interface_level', 'DEBUG')
```

1.8 Debugging Nipype Workflows

Throughout Nipype we try to provide meaningful error messages. If you run into an error that does not have a meaningful error message please let us know so that we can improve error reporting.
Here are some notes that may help debugging workflows or understanding performance issues.

1. Always run your workflow first on a single iterable (e.g. subject) and gradually increase the execution distribution complexity (Linear->MultiProc-> SGE).
2. Use the debug config mode. This can be done by setting:

   ```
   from nipype import config
   config.enable_debug_mode()
   ```

   as the first import of your nipype script.

   Note: Turning on debug will rerun your workflows and will rerun them after debugging is turned off.

3. There are several configuration options that can help with debugging. See *Configuration File* for more details:

   ```
   keep_inputs
   remove_unnecessary_outputs
   stop_on_first_crash
   stop_on_first_rerun
   ```

4. When running in distributed mode on cluster engines, it is possible for a node to fail without generating a crash file in the crashdump directory. In such cases, it will store a crash file in the *batch* directory.
5. All Nipype crashfiles can be inspected with the *nipype_display_crash* utility.
6. Nipype determines the hash of the input state of a node. If any input contains strings that represent files on the system path, the hash evaluation mechanism will determine the timestamp or content hash of each of those files. Thus any node with an input containing huge dictionaries (or lists) of file names can cause serious performance penalties.

7. For HUGE data processing, 'stop_on_first_crash':'False', is needed to get the bulk of processing done, and then 'stop_on_first_crash':'True', is needed for debugging and finding failing cases. Setting 'stop_on_first_crash': 'False' is a reasonable option when you would expect 90% of the data to execute properly.
8. Sometimes nipype will hang as if nothing is going on and if you hit Ctrl+C you will get a *ConcurrentLogHandler* error. Simply remove the pypeline.lock file in your home directory and continue.
9. One many clusters with shared NFS mounts synchronization of files across clusters may not happen before the typical NFS cache timeouts. When using PBS/LSF/SGE/Condor plugins in such cases the workflow may crash because it cannot retrieve the node result. Setting the *job_finished_timeout* can help: workflow.config['execution']['job_finished_timeout'] = 65

1.9 DataGrabber and DataSink explained

In this chapter we will try to explain the concepts behind DataGrabber and DataSink.

1.9.1 Why do we need these interfaces?

A typical workflow takes data as input and produces data as the result of one or more operations. One can set the data required by a workflow directly as illustrated below.

```
from fsl_tutorial2 import preproc
preproc.base_dir = os.path.abspath('.')
preproc.inputs.inputspec.func = os.path.abspath('data/s1/f3.nii')
preproc.inputs.inputspec.struct = os.path.abspath('data/s1/struct.nii')
preproc.run()
```

Typical neuroimaging studies require running workflows on multiple subjects or different parameterizations of algorithms. One simple approach to that would be to simply iterate over subjects.

```
from fsl_tutorial2 import preproc
for name in subjects:
  preproc.base_dir = os.path.abspath('.')
  preproc.inputs.inputspec.func = os.path.abspath('data/%s/f3.nii'%name)
  preproc.inputs.inputspec.struct = os.path.abspath('data/%s/struct.nii'%name)
  preproc.run()
```

However, in the context of complex workflows and given that users typically arrange their imaging and other data in a semantically hierarchical data store, an alternative mechanism for reading and writing the data generated by a workflow is often necessary. As the names suggest DataGrabber is used to get at data stored in a shared file system while DataSink is used to store the data generated by a workflow into a hierarchical structure on disk.

1.9.2 DataGrabber

DataGrabber is an interface for collecting files from hard drive. It is very flexible and supports almost any file organization of your data you can imagine.
You can use it as a trivial use case of getting a fixed file. By default, DataGrabber stores its outputs in a field called outfiles.

```
import nipype.interfaces.io as nio
datasource1 = nio.DataGrabber()
datasource1.inputs.base_directory = os.getcwd()
datasource1.inputs.template = 'data/s1/f3.nii'
results = datasource1.run()
```

Or you can get at all uncompressed NIfTI files starting with the letter 'f' in all directories starting with the letter 's'.

```
datasource2.inputs.base_directory = '/mass'
datasource2.inputs.template = 'data/s*/f*.nii'
```

Two special inputs were used in these previous cases. The input *base_directory* indicates in which directory to search, while the input *template* indicates the string template to match. So in the previous case datagrabber is looking for path matches of the form */mass/data/s*/f**.

Note: When used with wildcards (e.g., s* and f* above) DataGrabber does not return data in sorted order. In order to force it to return data in sorted order, one needs to set the input *sorted* = *True*. However, when explicitly specifying an order as we will see below, *sorted* should be set to *False*.

More useful cases arise when the template can be filled by other inputs. In the example below, we define an input field for *datagrabber* called *run*. This is then used to set the template (see %d in the template).

```
datasource3 = nio.DataGrabber(infields=['run'])
datasource3.inputs.base_directory = os.getcwd()
datasource3.inputs.template = 'data/s1/f%d.nii'
datasource3.inputs.run = [3, 7]
```

This will return files *basedir/data/s1/f3.nii* and *basedir/data/s1/f7.nii*. We can take this a step further and pair subjects with runs.

```
datasource4 = nio.DataGrabber(infields=['subject_id', 'run'])
datasource4.inputs.template = 'data/%s/f%d.nii'
datasource4.inputs.run = [3, 7]
datasource4.inputs.subject_id = ['s1', 's3']
```

This will return files *basedir/data/s1/f3.nii* and *basedir/data/s3/f7.nii*.

A more realistic use-case

In a typical study one often wants to grab different files for a given subject and store them in semantically meaningful outputs. In the following example, we wish to retrieve all the functional runs and the structural image for the subject 's1'.

```
datasource = nio.DataGrabber(infields=['subject_id'], outfields=['func', 'struct'])
datasource.inputs.base_directory = 'data'
datasource.inputs.template = '*'
datasource.inputs.field_template = dict(func='%s/f%d.nii',
                                        struct='%s/struct.nii')
datasource.inputs.template_args = dict(func=[['subject_id', [3,5,7,10]]],
                                       struct=[['subject_id']])
datasource.inputs.subject_id = 's1'
```

Two more fields are introduced: *field_template* and *template_args*. These fields are both dictionaries whose keys correspond to the *outfields* keyword. The *field_template* reflects the search path for each output field, while the *template_args* reflect the inputs that satisfy the template. The inputs can either be one of the named inputs specified by the *infields* keyword arg or it can be raw strings or integers corresponding to the template. For the *func* output, the **%s** in the *field_template* is satisfied by *subject_id* and the **%d** is field in by the list of numbers.

Note: We have not set *sorted* to *True* as we want the DataGrabber to return the functional files in the order it was specified rather than in an alphabetic sorted order.

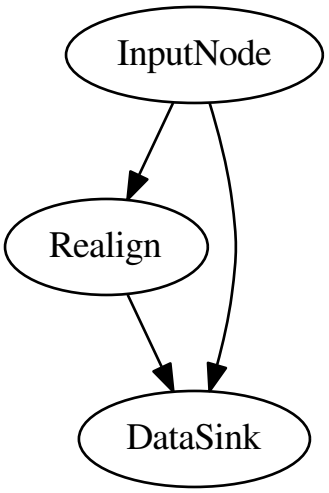
1.9.3 DataSink

A workflow working directory is like a **cache**. It contains not only the outputs of various processing stages, it also contains various extraneous information such as execution reports, hashfiles determining the input state of processes. All of this is embedded in a hierarchical structure that reflects the iterables that have been used in the workflow. This makes navigating the working directory a not so pleasant experience. And typically the user is

interested in preserving only a small percentage of these outputs. The DataSink interface can be used to extract components from this *cache* and store it at a different location. For XNAT-based storage, see XNATSink .

Note: Unlike other interfaces, a DataSink's inputs are defined and created by using the workflow connect statement. Currently disconnecting an input from the DataSink does not remove that connection port.

Let's assume we have the following workflow.



The following code segment defines the DataSink node and sets the *base_directory* in which all outputs will be stored. The *container* input creates a subdirectory within the *base_directory*. If you are iterating a workflow over subjects, it may be useful to save it within a folder with the subject id.

```
datasink = pe.Node(nio.DataSink(), name='sinker')
datasink.inputs.base_directory = '/path/to/output'
workflow.connect(inputnode, 'subject_id', datasink, 'container')
```

If we wanted to save the realigned files and the realignment parameters to the same place the most intuitive option would be:

```
workflow.connect(realigner, 'realigned_files', datasink, 'motion')
workflow.connect(realigner, 'realignment_parameters', datasink, 'motion')
```

However, this will not work as only one connection is allowed per input port. So we need to create a second port. We can store the files in a separate folder.

```
workflow.connect(realigner, 'realigned_files', datasink, 'motion')
workflow.connect(realigner, 'realignment_parameters', datasink, 'motion.par')
```

The period (.) indicates that a subfolder called par should be created. But if we wanted to store it in the same folder as the realigned files, we would use the .@ syntax. The @ tells the DataSink interface to not create the subfolder. This will allow us to create different named input ports for DataSink and allow the user to store the files in the same folder.

```
workflow.connect(realigner, 'realigned_files', datasink, 'motion')
workflow.connect(realigner, 'realignment_parameters', datasink, 'motion.@par')
```

The syntax for the input port of DataSink takes the following form:

```
string[[.[@]]string[[.[@]]string] ...]
where parts between paired [] are optional.
```

MapNode

In order to use DataSink inside a MapNode, it's inputs have to be defined inside the constructor using the *infields* keyword arg.

Parameterization

As discussed in MapNode, iterfield, and iterables explained, one can run a workflow iterating over various inputs using the iterables attribute of nodes. This means that a given workflow can have multiple outputs depending on how many iterables are there. Iterables create working directory subfolders such as *_iterable_name_value*. The *parameterization* input parameter controls whether the data stored using DataSink is in a folder structure that contains this iterable information or not. It is generally recommended to set this to *True* when using multiple nested iterables.

Substitutions

The *substitutions* and *substitutions_regexp* inputs allow users to modify the output destination path and name of a file. Substitutions are a list of 2-tuples and are carried out in the order in which they were entered. Assuming that the output path of a file is:

```
/root/container/_variable_1/file_subject_realigned.nii
```

we can use substitutions to clean up the output path.

```
datasink.inputs.substitutions = [('_variable', 'variable'),
                                 ('file_subject_', '')]
```

This will rewrite the file as:

```
/root/container/variable_1/realigned.nii
```

Note: In order to figure out which substitutions are needed it is often useful to run the workflow on a limited set of iterables and then determine the substitutions.

1.10 The SelectFiles Interfaces

Nipype 0.9 introduces a new interface for intelligently finding files on the disk and feeding them into your workflows: SelectFiles. SelectFiles is intended as a simpler alternative to the DataGrabber interface that was discussed previously in DataGrabber and DataSink explained.
SelectFiles is built on Python format strings, which are similar to the Python string interpolation feature you are likely already familiar with, but advantageous in several respects. Format strings allow you to replace named sections of template strings set off by curly braces (*{}*), possibly filtered through a set of functions that control how the values are rendered into the string. As a very basic example, we could write

```
msg = "This workflow uses {package}"
```

and then format it with keyword arguments:

```
print msg.format(package="FSL")
```

SelectFiles only requires that you provide templates that can be used to find your data; the actual formatting happens behind the scenes.
Consider a basic example in which you want to select a T1 image and multple functional images for a number of subjects. Invoking SelectFiles in this case is quite straightforward:

```
from nipype import SelectFiles
templates = dict(T1="data/{subject_id}/struct/T1.nii",
                 epi="data/{subject_id}/func/epi_run*.nii")
sf = SelectFiles(templates)
```

SelectFiles will take the *templates* dictionary and parse it to determine its own inputs and oututs. Specifically, each name used in the format spec (here just *subject_id*) will become an interface input, and each key in the dictionary (here *T1* and *epi*) will become interface outputs. The *templates* dictionary thus succinctly links the node inputs to the appropriate outputs. You'll also note that, as was the case with DataGrabber, you can use basic glob syntax to match multiple files for a given output field. Additionally, any of the conversions outlined in the Python documentation for format strings can be used in the templates.

There are a few other options that help make SelectFiles flexible enough to deal with any situation where you need to collect data. Like DataGrabber, SelectFiles has a *base_directory* parameter that allows you to specify a path that is common to all of the values in the *templates* dictionary. Additionally, as *glob* does not return a sorted list, there is also a *sort_filelist* option, taking a boolean, to control whether sorting should be applied (it is True by default).

The final input is *force_lists*, which controls how SelectFiles behaves in cases where only a single file matches the template. The default behavior is that when a template matches multiple files they are returned as a list, while a single file is returned as a string. There may be situations where you want to force the outputs to always be returned as a list (for example, you are writing a workflow that expects to operate on several runs of data, but some of your subjects only have a single run). In this case, *force_lists* can be used to tune the outputs of the interface. You can either use a boolean value, which will be applied to every output the interface has, or you can provide a list of the output fields that should be coerced to a list. Returning to our basic example, you may want to ensure that the *epi* files are returned as a list, but you only ever will have a single *T1* file. In this case, you would do

```
sf = SelectFiles(templates, force_lists=["epi"])
```

1.11 The Function Interface

Most Nipype interfaces provide access to external programs, such as FSL binaries or SPM routines. However, a special interface, `nipype.interfaces.utility.Function`, allows you to wrap arbitrary Python code in the Interface framework and seamlessly integrate it into your workflows.

1.11.1 A Simple Function Interface

The most important component of a working Function interface is a Python function. There are several ways to associate a function with a Function interface, but the most common way will involve functions you code yourself as part of your Nipype scripts. Consider the following function:

```
def add_two(val):
    return val + 2
```

This simple function takes a value, adds 2 to it, and returns that new value.

Just as Nipype interfaces have inputs and outputs, Python functions have inputs, in the form of parameters or arguments, and outputs, in the form of their return values. When you define a Function interface object with an existing function, as in the case of `add_two()` above, you must pass the constructor information about the function's inputs, its outputs, and the function itself. For example,

```
from nipype.interfaces.utility import Function
add_two_interface = Function(input_names=["val"],
                             output_names=["out_val"],
                             function=add_two)
```

Then you can set the inputs and run just as you would with any other interface:

```
add_two_interface.inputs.val = 2
res = add_two_interface.run()
print res.outputs.out_val
```

Which would print `4`.
Note that, if you are working interactively, the Function interface is unable to use functions that are defined within your interpreter session. (Specifically, it can't use functions that live in the `__main__` namespace).

1.11.2 Using External Packages

Chances are, you will want to write functions that do more complicated processing, particularly using the growing stack of Python packages geared towards neuroimaging, such as Nibabel, Nipy, or PyMVPA.
While this is completely possible (and, indeed, an intended use of the Function interface), it does come with one important constraint. The function code you write is executed in a standalone environment, which means that any external functions or classes you use have to be imported within the function itself:

```
def get_n_trs(in_file):
    import nibabel
    f = nibabel.load(in_file)
    return f.shape[-1]
```

Without explicitly importing Nibabel in the body of the function, this would fail.
Alternatively, it is possible to provide a list of strings corresponding to the imports needed to execute a function as a parameter of the *Function* constructor. This allows for the use of external functions that do not import all external definitions inside the function body.

1.11.3 Hello World - Function interface in a workflow

Contributed by: Hänel Nikolaus Valentin
The following snippet of code demonstrates the use of the function interface in the context of a workflow. Note the use of `import os` within the function as well as returning the absolute path from the Hello function. The *import* inside is necessary because functions are coded as strings and do not have to be on the PYTHONPATH. However any function called by this function has to be available on the PYTHONPATH. The *absolute path* is necessary because all workflow nodes are executed in their own directory and therefore there is no way of determining that the input file came from a different directory:

```
import nipype.pipeline.engine as pe
from nipype.interfaces.utility import Function

def Hello():
   import os
   from nipype import logging
   iflogger = logging.getLogger('interface')
   message = "Hello "
   file_name =  'hello.txt'
   iflogger.info(message)
   with open(file_name, 'w') as fp:
       fp.write(message)
   return os.path.abspath(file_name)

def World(in_file):
   from nipype import logging
   iflogger = logging.getLogger('interface')
   message = "World!"
   iflogger.info(message)
   with open(in_file, 'a') as fp:
       fp.write(message)
```

```
    hello = pe.Node(name='hello',
                    interface=Function(input_names=[],
                                       output_names=['out_file'],
                                       function=Hello))
    world = pe.Node(name='world',
                    interface=Function(input_names=['in_file'],
                                       output_names=[],
                                       function=World))

    pipeline = pe.Workflow(name='nipype_demo')
    pipeline.connect([(hello, world, [('out_file', 'in_file')])])
    pipeline.run()
    pipeline.write_graph(graph2use='flat')
```

1.11.4 Advanced Use

To use an existing function object (as we have been doing so far) with a Function interface, it must be passed to the constructor. However, it is also possible to dynamically set how a Function interface will process its inputs using the special `function_str` input.
This input takes not a function object, but actually a single string that can be parsed to define a function. In the equivalent case to our example above, the string would be

```
    add_two_str = "def add_two(val):\n    return val + 2\n"
```

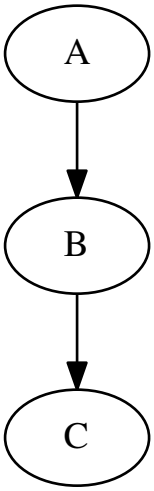
Unlike when using a function object, this input can be set like any other, meaning that you could write a function that outputs different function strings depending on some run-time contingencies, and connect that output the the `function_str` input of a downstream Function interface.

1.12 MapNode, iterfield, and iterables explained

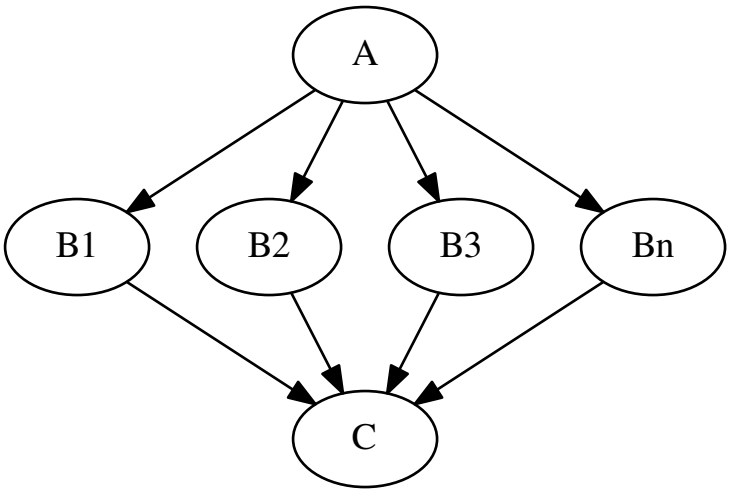
In this chapter we will try to explain the concepts behind MapNode, iterfield, and iterables.

1.12.1 MapNode and iterfield

Imagine that you have a list of items (lets say files) and you want to execute the same node on them (for example some smoothing or masking). Some nodes accept multiple files and do exactly the same thing on them, but some don't (they expect only one file). MapNode can solve this problem. Imagine you have the following workflow:



Node "A" outputs a list of files, but node "B" accepts only one file. Additionally "C" expects a list of files. What you would like is to run "B" for every file in the output of "A" and collect the results as a list and feed it to "C". Something like this:



The code to achieve this is quite simple

```
import nipype.pipeline.engine as pe
a = pe.Node(interface=A(), name="a")
b = pe.MapNode(interface=B(), name="b", iterfield=['in_file'])
c = pe.Node(interface=C(), name="c")

my_workflow = pe.Workflow(name="my_workflow")
```

```
my_workflow.connect([(a,b,[('out_files','in_file')]),
                     (b,c,[('out_file','in_files')])
                     ])
```

assuming that interfaces "A" and "C" have one input "in_files" and one output "out_files" (both lists of files). Interface "B" has single file input "in_file" and single file output "out_file".

You probably noticed that you connect nodes as if "B" could accept and output list of files. This is because it is wrapped using MapNode instead of Node. This special version of node will (under the bonnet) create an instance of "B" for every item in the list from the input. The compulsory argument "iterfield" defines which input should it iterate over (for example in single file smooth interface you would like to iterate over input files not the smoothing width). At the end outputs are collected into a list again. In other words this is map and reduce scenario.

You might have also noticed that the iterfield arguments expects a list of input names instead of just one name. This suggests that there can be more than one! Even though a bit confusing this is true. You can specify more than one input to iterate over but the lists that you provide (for all the inputs specified in iterfield) have to have the same length. MapNode will then pair the parameters up and run the first instance with first set of parameters and second with second set of parameters. For example, this code:

```
b = pe.MapNode(interface=B(), name="b", iterfield=['in_file', 'n'])
b.inputs.in_file = ['file', 'another_file', 'different_file']
b.inputs.n = [1,2,3]
b.run()
```

is almost the same as running

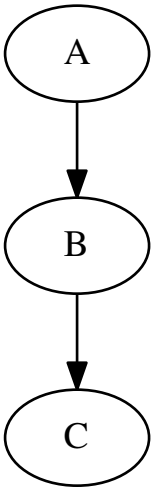
```
b1 = pe.Node(interface=B(), name="b1")
b1.inputs.in_file = 'file'
b1.inputs.n = 1

b2 = pe.Node(interface=B(), name="b2")
b2.inputs.in_file = 'another_file'
b2.inputs.n = 2

b3 = pe.Node(interface=B(), name="b3")
b3.inputs.in_file = 'different_file'
b3.inputs.n = 3
```

It is a rarely used feature, but you can sometimes find it useful.

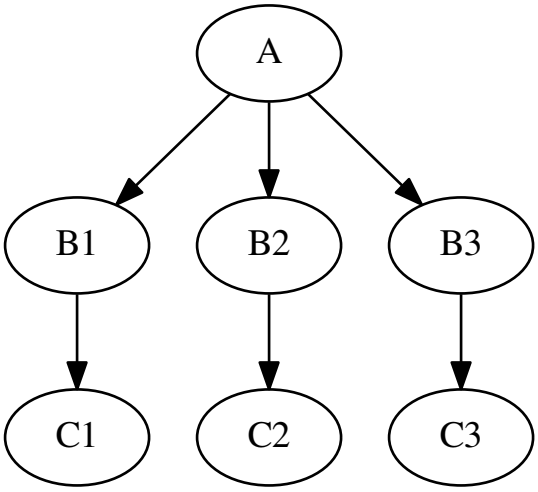
In more advanced applications it is useful to be able to iterate over items of nested lists (for example [[1,2],[3,4]]). MapNode allows you to do this with the "nested=True" parameter. Outputs will preserve the same nested structure as the inputs.

1.12.2 Iterables

Now imagine a different scenario. You have your workflow as before



and there are three possible values of one of the inputs node "B" you would like to investigate (for example width of 2,4, and 6 pixels of a smoothing node). You would like to see how different parameters in node "B" would influence everything that depends on its outputs (node "C" in our example). Therefore the new graph should look like this:



Of course you can do it manually by creating copies of all the nodes for different parameter set, but this can be very time consuming, especially when there are more than one node taking inputs from "B". Luckily nipype supports this scenario! Its called iterables and and you use it this way:

```
import nipype.pipeline.engine as pe
a = pe.Node(interface=A(), name="a")
b = pe.Node(interface=B(), name="b")
```

```
b.iterables = ("n", [1, 2, 3])
c = pe.Node(interface=C(), name="c")

my_workflow = pe.Workflow(name="my_workflow")
my_workflow.connect([(a,b,[('out_file','in_file')]),
                     (b,c,[('out_file','in_file')])
                     ])
```

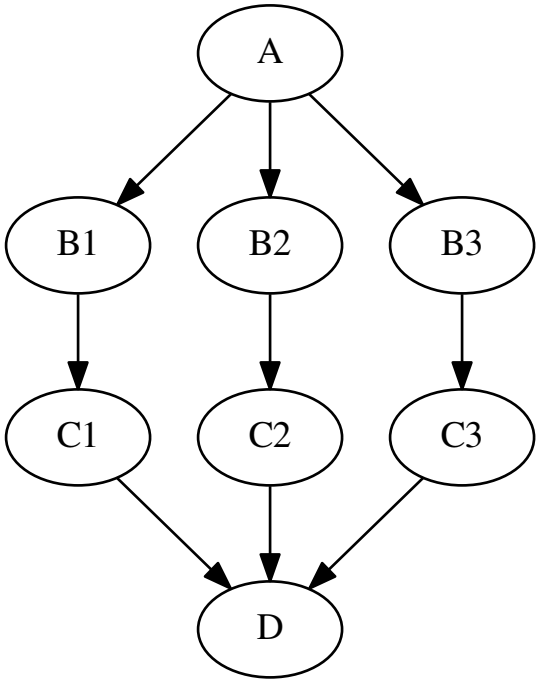
Assuming that you want to try out values 1, 2, and 3 of input "n" of the node "B". This will also create three different versions of node "C" - each with inputs from instances of node "C" with different values of "n". Additionally, you can set multiple iterables for a node with a list of tuples in the above format.
Iterables are commonly used to execute the same workflow for many subjects. Usually one parametrises DataGrabber node with subject ID. This is achieved by connecting an IdentityInterface in front of DataGrabber. When you set iterables of the IdentityInterface to the list of subjects IDs, the same workflow will be executed for every subject. See `examples/fmri_spm` to see this pattern in action.

1.13 JoinNode, synchronize and itersource

The previous MapNode, iterfield, and iterables explained chapter described how to fork and join nodes using MapNode and iterables. In this chapter, we introduce features which build on these concepts to add workflow flexibility.

1.13.1 JoinNode, joinsource and joinfield

A `nipype.pipeline.engine.JoinNode` generalizes MapNode to operate in conjunction with an upstream iterable node to reassemble downstream results, e.g.:



The code to achieve this is as follows:

```
import nipype.pipeline.engine as pe
a = pe.Node(interface=A(), name="a")
b = pe.Node(interface=B(), name="b")
b.iterables = ("in_file", images)
c = pe.Node(interface=C(), name="c")
d = pe.JoinNode(interface=D(), joinsource="b",
                joinfield="in_files", name="d")

my_workflow = pe.Workflow(name="my_workflow")
my_workflow.connect([(a,b,[('subject','subject')]),
                     (b,c,[('out_file','in_file')])
                     (c,d,[('out_file','in_files')])
                     ])
```

This example assumes that interface "A" has one output *subject*, interface "B" has two inputs *subject* and *in_file* and one output *out_file*, interface "C" has one input *in_file* and one output *out_file*, and interface D has one list input *in_files*. The *images* variable is a list of three input image file names.
As with *iterables* and the MapNode *iterfield*, the *joinfield* can be a list of fields. Thus, the declaration in the previous example is equivalent to the following:

```
d = pe.JoinNode(interface=D(), joinsource="b",
                joinfield=["in_files"], name="d")
```

The *joinfield* defaults to all of the JoinNode input fields, so the declaration is also equivalent to the following:

```
d = pe.JoinNode(interface=D(), joinsource="b", name="d")
```

In this example, the node "c" *out_file* outputs are collected into the JoinNode "d" *in_files* input list. The *in_files* order is the same as the upstream "b" node iterables order.
The JoinNode input can be filtered for unique values by specifying the *unique* flag, e.g.:

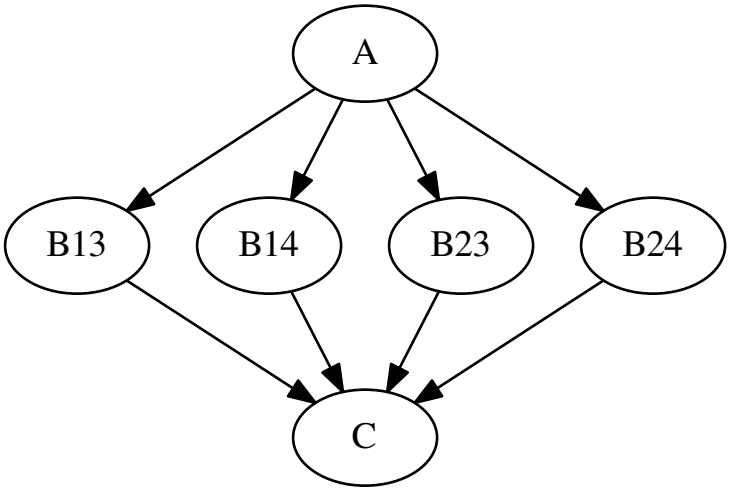
```
d = pe.JoinNode(interface=D(), joinsource="b", unique=True, name="d")
```

1.13.2 synchronize

The `nipype.pipeline.engine.Node` *iterables* parameter can be be a single field or a list of fields. If it is a list, then execution is performed over all permutations of the list items. For example:

```
b.iterables = [("m", [1, 2]), ("n", [3, 4])]
```

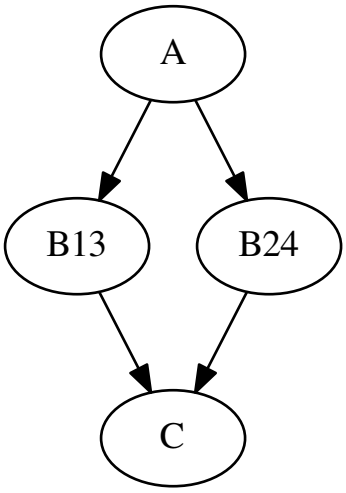
results in the execution graph:



where "B13" has inputs $m = 1$, $n = 3$, "B14" has inputs $m = 1$, $n = 4$, etc.
The *synchronize* parameter synchronizes the iterables lists, e.g.:

```
b.iterables = [("m", [1, 2]), ("n", [3, 4])]
b.synchronize = True
```

results in the execution graph:



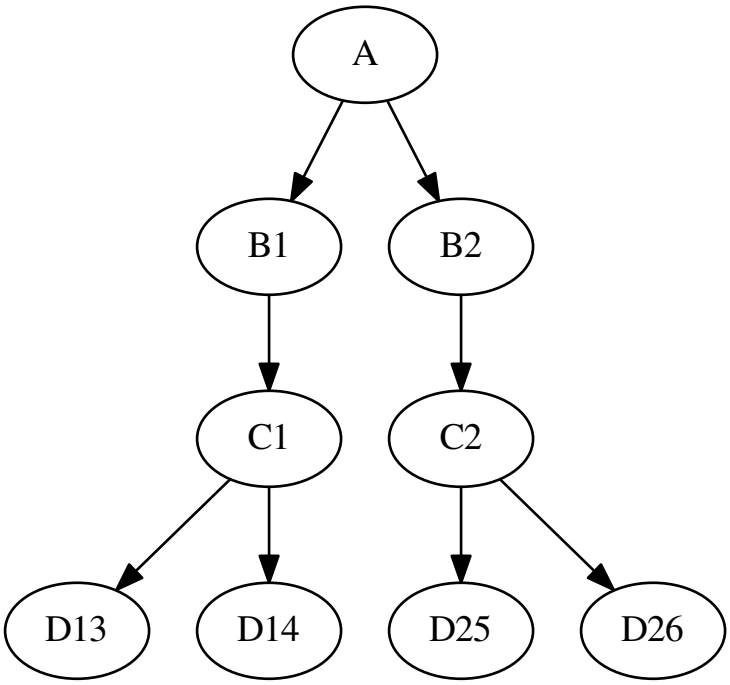
where the iterable inputs are selected in lock-step by index, i.e.:
$(m, n) = (1, 3)$ and $(2, 4)$
for "B13" and "B24", resp.

1.13.3 itersource

The *itersource* feature allows you to expand a downstream iterable based on a mapping of an upstream iterable. For example:

```
a = pe.Node(interface=A(), name="a")
b = pe.Node(interface=B(), name="b")
b.iterables = ("m", [1, 2])
c = pe.Node(interface=C(), name="c")
d = pe.Node(interface=D(), name="d")
d.itersource = ("b", "m")
d.iterables = [("n", {1:[3,4], 2:[5,6]})]
my_workflow = pe.Workflow(name="my_workflow")
my_workflow.connect([(a,b,[('out_file','in_file')]),
                     (b,c,[('out_file','in_file')])
                     (c,d,[('out_file','in_file')])
                     ])
```

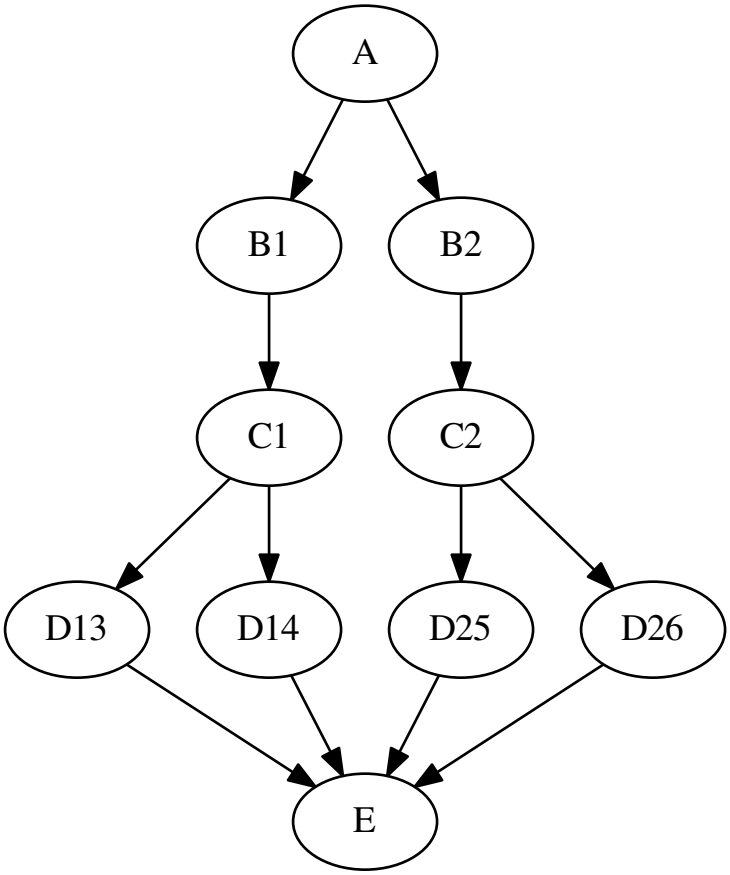
results in the execution graph:



In this example, all interfaces have input *in_file* and output *out_file*. In addition, interface "B" has input *m* and interface "D" has input *n*. A Python dictionary associates the "b" node input value with the downstream "d" node *n* iterable values.
This example can be extended with a summary JoinNode:

```
e = pe.JoinNode(interface=E(), joinsource="d",
                joinfield="in_files", name="e")
my_workflow.connect(d, 'out_file',
                    e, 'in_files')
```

resulting in the graph:



The combination of iterables, MapNode, JoinNode, synchronize and itersource enables the creation of arbitrarily complex workflow graphs. The astute workflow builder will recognize that this flexibility is both a blessing and a curse. These advanced features are handy additions to the Nipype toolkit when used judiciously.

1.14 Model Specification for First Level fMRI Analysis

Nipype provides a general purpose model specification mechanism with specialized subclasses for package specific extensions.

1.14.1 General purpose model specification

The `SpecifyModel` provides a generic mechanism for model specification. A mandatory input called subject_info provides paradigm specification for each run corresponding to a subject. This has to be in the form of a `Bunch` or a list of Bunch objects (one for each run). Each Bunch object contains the following attribules.

Required for most designs

- conditions : list of names
- onsets : lists of onsets corresponding to each condition

- **durations** [lists of durations corresponding to each condition. Should be] left to a single 0 if all events are being modelled as impulses.

Optional

- regressor_names : list of names corresponding to each column. Should be None if automatically assigned.
- regressors : list of lists. values for each regressor - must correspond to the number of volumes in the functional run
- **amplitudes** [lists of amplitudes for each event. This will be ignored by] SPM's Level1Design.

The following two (tmod, pmod) will be ignored by any Level1Design class other than SPM:

- **tmod** [lists of conditions that should be temporally modulated. Should] default to None if not being used.
- **pmod** [list of Bunch corresponding to conditions]
  name : name of parametric modulator
  param : values of the modulator
  poly : degree of modulation

An example Bunch definition:

```
from nipype.interfaces.base import Bunch
condnames = ['Tapping', 'Speaking', 'Yawning']
event_onsets = [[0, 10, 50], [20, 60, 80], [30, 40, 70]]
durations = [[0],[0],[0]]

subject_info = Bunch(conditions=condnames,
                                    onsets = event_onsets,
                                    durations = durations)
```

Alternatively, you can provide condition, onset, duration and amplitude information through event files. The event files have to be in 1,2 or 3 column format with the columns corresponding to Onsets, Durations and Amplitudes and they have to have the name event_name.run<anything else> e.g.: Words.run001.txt. The event_name part will be used to create the condition names. Words.run001.txt may look like:

```
# Word Onsets Durations
 0   10
 20   10
 ...
```

or with amplitudes:

```
# Word Onsets Durations Amplitudes
0    10     1
20   10    1
...
```

Together with this information, one needs to specify:

- whether the durations and event onsets are specified in terms of scan volumes or secs.
- the high-pass filter cutoff,
- the repetition time per scan
- functional data files corresponding to each run.

Optionally you can specify realignment parameters, outlier indices. Outlier files should contain a list of numbers, one per row indicating which scans should not be included in the analysis. The numbers are 0-based.

1.14.2 SPM specific attributes

in addition to the generic specification options, several SPM specific options can be provided. In particular, the subject_info function can provide temporal and parametric modulators in the Bunch attributes tmod and pmod. The following example adds a linear parametric modulator for speaking rate for the events specified earlier:

```
pmod = [None, Bunch(name=['Rate'], param=[[.300, .500, .600]],
                                    poly=[1]), None]
```

```
subject_info = Bunch(conditions=condnames,
                                    onsets = event_onsets,
                                    durations = durations,
                                    pmod = pmod)
```

`SpecifySPMModel` also allows specifying additional components. If you have a study with multiple runs, you can choose to concatenate conditions from different runs. by setting the input option **concatenate_runs** to True. You can also choose to set the output options for this class to be in terms of 'scans'.

1.14.3 Sparse model specification

In addition to standard models, `SpecifySparseModel` allows model generation for sparse and sparse-clustered acquisition experiments. Details of the model generation and utility are provided in Ghosh et al. (2009) OHBM 2009.

1.15 Saving Workflows and Nodes to a file (experimental)

On top of the standard way of saving (i.e. serializing) objects in Python (see pickle) Nipype provides methods to turn Workflows and nodes into human readable code. This is useful if you want to save a Workflow that you have generated on the fly for future use.
To generate Python code for a Workflow use the export method:

```
from nipype.interfaces.fsl import BET, ImageMaths
from nipype.pipeline.engine import Workflow, Node, MapNode, format_node
from nipype.interfaces.utility import Function, IdentityInterface

bet = Node(BET(), name='bet')
bet.iterables = ('frac', [0.3, 0.4])

bet2 = MapNode(BET(), name='bet2', iterfield=['infile'])
bet2.iterables = ('frac', [0.4, 0.5])

maths = Node(ImageMaths(), name='maths')

def testfunc(in1):
    """dummy func
    """
    out = in1 + 'foo' + "out1"
    return out

funcnode = Node(Function(input_names=['a'], output_names=['output'], function=testfunc),
                name='testfunc')
funcnode.inputs.in1 = '-sub'
func = lambda x: x

inode = Node(IdentityInterface(fields=['a']), name='inode')

wf = Workflow('testsave')
wf.add_nodes([bet2])
wf.connect(bet, 'mask_file', maths, 'in_file')
wf.connect(bet2, ('mask_file', func), maths, 'in_file2')
wf.connect(inode, 'a', funcnode, 'in1')
wf.connect(funcnode, 'output', maths, 'op_string')

wf.export()
```

This will create a file "outputtestsave.py" with the following content:

```
from nipype.pipeline.engine import Workflow, Node, MapNode
from nipype.interfaces.utility import IdentityInterface
from nipype.interfaces.utility import Function
from nipype.utils.misc import getsource
from nipype.interfaces.fsl.preprocess import BET
from nipype.interfaces.fsl.utils import ImageMaths
# Functions
func = lambda x: x
# Workflow
testsave = Workflow("testsave")
# Node: testsave.inode
inode = Node(IdentityInterface(fields=['a'], mandatory_inputs=True), name="inode")
# Node: testsave.testfunc
testfunc = Node(Function(input_names=['a'], output_names=['output']), name="testfunc")
def testfunc_1(in1):
    """dummy func
    """
    out = in1 + 'foo' + "out1"
    return out

testfunc.inputs.function_str = getsource(testfunc_1)
testfunc.inputs.ignore_exception = False
testfunc.inputs.in1 = '-sub'
testsave.connect(inode, "a", testfunc, "in1")
# Node: testsave.bet2
bet2 = MapNode(BET(), iterfield=['infile'], name="bet2")
bet2.iterables = ('frac', [0.4, 0.5])
bet2.inputs.environ = {'FSLOUTPUTTYPE': 'NIFTI_GZ'}
bet2.inputs.ignore_exception = False
bet2.inputs.output_type = 'NIFTI_GZ'
bet2.inputs.terminal_output = 'stream'
# Node: testsave.bet
bet = Node(BET(), name="bet")
bet.iterables = ('frac', [0.3, 0.4])
bet.inputs.environ = {'FSLOUTPUTTYPE': 'NIFTI_GZ'}
bet.inputs.ignore_exception = False
bet.inputs.output_type = 'NIFTI_GZ'
bet.inputs.terminal_output = 'stream'
# Node: testsave.maths
maths = Node(ImageMaths(), name="maths")
maths.inputs.environ = {'FSLOUTPUTTYPE': 'NIFTI_GZ'}
maths.inputs.ignore_exception = False
maths.inputs.output_type = 'NIFTI_GZ'
maths.inputs.terminal_output = 'stream'
testsave.connect(bet2, ('mask_file', func), maths, "in_file2")
testsave.connect(bet, "mask_file", maths, "in_file")
testsave.connect(testfunc, "output", maths, "op_string")
```

The file is ready to use and includes all the necessary imports.

1.16 Using SPM with MATLAB Common Runtime

In order to use the standalone MCR version of spm, you need to ensure that the following commands are executed at the beginning of your script:

```
from nipype.interfaces import spm
matlab_cmd = '/path/to/run_spm8.sh /path/to/Compiler_Runtime/v713/ script'
spm.SPMCommand.set_mlab_paths(matlab_cmd=matlab_cmd, use_mcr=True)
```

you can test by calling:

```
spm.SPMCommand().version
```

If you want to enforce the standalone MCR version of spm for nipype globally, you can do so by setting the following environment variables:

SPMMCRCMD Specifies the command to use to run the spm standalone MCR version. You may still override the command as described above.

FORCE_SPMMCR Set this to any value in order to enforce the use of spm standalone MCR version in nipype globally. Technically, this sets the *use_mcr* flag of the spm interface to True.

Information about the MCR version of SPM8 can be found at:
http://en.wikibooks.org/wiki/SPM/Standalone

1.17 Using MIPAV, JIST, and CBS Tools

If you are trying to use MIPAV, JIST or CBS Tools interfaces you need to configure CLASSPATH environmental variable correctly. It needs to include extensions shipped with MIPAV, MIPAV itself and MIPAV plugins. For example:

In order to use the standalone MCR version of spm, you need to ensure that the following commands are executed at the beginning of your script:

```
# location of additional JAVA libraries to use
JAVALIB=/Applications/mipav/jre/Contents/Home/lib/ext/

# location of the MIPAV installation to use
MIPAV=/Applications/mipav
# location of the plugin installation to use
# please replace 'ThisUser' by your user name
PLUGINS=/Users/ThisUser/mipav/plugins

export CLASSPATH=$JAVALIB/*:$MIPAV:$MIPAV/lib/*:$PLUGINS
```

1.18 Running Nipype Interfaces from the command line (nipype_cmd)

The primary use of Nipype is to build automated non-interactive pipelines. However, sometimes there is a need to run some interfaces quickly from the command line. This is especially useful when running Interfaces wrapping code that does not have command line equivalents (nipy or SPM). Being able to run Nipype interfaces opens new possibilities such as inclusion of SPM processing steps in bash scripts.

To run Nipype Interafces you need to use the nipype_cmd tool that should already be installed. The tool allows you to list Interfaces available in a certain package:

```
$nipype_cmd nipype.interfaces.nipy

Available Interfaces:
    SpaceTimeRealigner
    Similarity
    ComputeMask
    FitGLM
    EstimateContrast
    FmriRealign4d
```

After selecting a particular Interface you can learn what inputs it requires:

```
$nipype_cmd nipype.interfaces.nipy ComputeMask --help
```

```
usage:nipype_cmd nipype.interfaces.nipy ComputeMask [-h] [--M M] [--cc CC]
                                                     [--ignore_exception IGNORE_EXCEPTION]
                                                     [--m M]
                                                     [--reference_volume REFERENCE_VOLUME]
                                                     mean_volume

Run ComputeMask

positional arguments:
  mean_volume           mean EPI image, used to compute the threshold for the
                        mask

optional arguments:
  -h, --help            show this help message and exit
  --M M                 upper fraction of the histogram to be discarded
  --cc CC               Keep only the largest connected component
  --ignore_exception IGNORE_EXCEPTION
                        Print an error message instead of throwing an
                        exception in case the interface fails to run
  --m M                 lower fraction of the histogram to be discarded
  --reference_volume REFERENCE_VOLUME
                        reference volume used to compute the mask. If none is
                        give, the mean volume is used.
```

Finally you can run run the Interface:

```
$nipype_cmd nipype.interfaces.nipy ComputeMask mean.nii.gz
```

All that from the command line without having to start python interpreter manually.

CHAPTER 2

Changes in Nipype

2.1 Release 0.11.0 (September 15, 2015)

- API: Change how hash values are computed (https://github.com/nipy/nipype/pull/1174)
- **ENH: New algorithm: mesh.WarpPoints applies displacements fields to point sets** (https://github.com/nipy/nipype/pull/889).
- ENH: New interfaces for MRTrix3 (https://github.com/nipy/nipype/pull/1126)
- ENH: New option in afni.3dRefit - zdel, ydel, zdel etc. (https://github.com/nipy/nipype/pull/1079)
- FIX: ants.Registration composite transform outputs are no longer returned as lists (https://github.com/nipy/nipype/pull/1183)
- **BUG: ANTs Registration interface failed with multi-modal inputs** (https://github.com/nipy/nipype/pull/1176) (https://github.com/nipy/nipype/issues/1175)
- ENH: dipy.TrackDensityMap interface now accepts a reference image (https://github.com/nipy/nipype/pull/1091)
- FIX: Bug in XFibres5 (https://github.com/nipy/nipype/pull/1168)
- **ENH: Attempt to use hard links for data sink.** (https://github.com/nipy/nipype/pull/1161)
- **FIX: Updates to SGE Plugins** (https://github.com/nipy/nipype/pull/1129)
- **ENH: Add ants JointFusion() node with testing** (https://github.com/nipy/nipype/pull/1160)
- **ENH: Add –float option for antsRegistration calls** (https://github.com/nipy/nipype/pull/1159)
- **ENH: Added interface to simulate DWIs using the multi-tensor model** (https://github.com/nipy/nipype/pull/1085)
- ENH: New interface for FSL fslcpgeom utility (https://github.com/nipy/nipype/pull/1152)
- ENH: Added SLURMGraph plugin for submitting jobs to SLURM with dependencies (https://github.com/nipy/nipype/pull/1136)
- **FIX: Enable absolute path definitions in DCMStack (https://github.com/nipy/nipype/pull/1089,** replaced by https://github.com/nipy/nipype/pull/1093)
- **ENH: New mesh.MeshWarpMaths to operate on surface-defined warpings** (https://github.com/nipy/nipype/pull/1016)
- **FIX: Refactor P2PDistance, change name to ComputeMeshWarp, add regression tests,** fix bug in area weighted distance, and added optimizations (https://github.com/nipy/nipype/pull/1016)
- ENH: Add an option not to resubmit Nodes that finished running when using SGEGraph (https://github.com/nipy/nipype/pull/1002)
- FIX: FUGUE is now properly listing outputs. (https://github.com/nipy/nipype/pull/978)
- **ENH: Improved FieldMap-Based (FMB) workflow for correction of susceptibility distortions in EPI seqs.** (https://github.com/nipy/nipype/pull/1019)
- FIX: In the FSLXcommand _list_outputs function fixed for loop range (https://github.com/nipy/nipype/pull/1071)
- ENH: Dropped support for now 7 years old Python 2.6 (https://github.com/nipy/nipype/pull/1069)
- FIX: terminal_output is not mandatory anymore (https://github.com/nipy/nipype/pull/1070)
- ENH: Added “nipype_cmd” tool for running interfaces from the command line (https://github.com/nipy/nipype/pull/795)
- FIX: Fixed Camino output naming (https://github.com/nipy/nipype/pull/1061)
- ENH: Add the average distance to ErrorMap (https://github.com/nipy/nipype/pull/1039)

- ENH: Inputs with name_source can be now chained in cascade (https://github.com/nipy/nipype/pull/938)
- **ENH: Improve JSON interfaces: default settings when reading and consistent output creation** when writing (https://github.com/nipy/nipype/pull/1047)
- FIX: AddCSVRow problems when using infields (https://github.com/nipy/nipype/pull/1028)
- FIX: Removed unused ANTS registration flag (https://github.com/nipy/nipype/pull/999)
- FIX: Amend create_tbss_non_fa() workflow to match FSL's tbss_non_fa command. (https://github.com/nipy/nipype/pull/1033)
- FIX: remove unused mandatory flag from spm normalize (https://github.com/nipy/nipype/pull/1048)
- ENH: Update ANTSCorticalThickness interface (https://github.com/nipy/nipype/pull/1013)
- FIX: Edge case with sparsemodels and PEP8 cleanup (https://github.com/nipy/nipype/pull/1046)
- ENH: New io interfaces for JSON files reading/writing (https://github.com/nipy/nipype/pull/1020)
- ENH: Enhanced openfmri script to support freesurfer linkage (https://github.com/nipy/nipype/pull/1037)
- BUG: matplotlib is supposed to be optional (https://github.com/nipy/nipype/pull/1003)
- FIX: Fix split_filename behaviour when path has no file component (https://github.com/nipy/nipype/pull/1035)
- ENH: Updated FSL dtifit to include option for grad non-linearities (https://github.com/nipy/nipype/pull/1032)
- **ENH: Updated Camino tracking interfaces, which can now use FSL bedpostx output.** New options also include choice of tracker, interpolator, stepsize and curveinterval for angle threshold (https://github.com/nipy/nipype/pull/1029)
- FIX: Interfaces redirecting X crashed if $DISPLAY not defined (https://github.com/nipy/nipype/pull/1027)
- FIX: Bug crashed 'make api' (https://github.com/nipy/nipype/pull/1026)
- ENH: Updated antsIntroduction to handle RA and RI registrations (https://github.com/nipy/nipype/pull/1009)
- **ENH: Updated N4BiasCorrection input spec to include weight image and spline order. Made** argument formatting consistent. Cleaned ants.segmentation according to PEP8. (https://github.com/nipy/nipype/pull/990/files)
- ENH: SPM12 Normalize interface (https://github.com/nipy/nipype/pull/986)
- FIX: Utility interface test dir (https://github.com/nipy/nipype/pull/986)
- FIX: IPython engine directory reset after crash (https://github.com/nipy/nipype/pull/987)
- ENH: Resting state fMRI example with NiPy realignment and no SPM (https://github.com/nipy/nipype/pull/992)
- **FIX: Corrected Freesurfer SegStats _list_outputs to avoid error if summary_file is** undefined (issue #994)(https://https://github.com/nipy/nipype/pull/996)
- FIX: OpenfMRI support and FSL 5.0.7 changes (https://github.com/nipy/nipype/pull/1006)
- FIX: Output prefix in SPM Normalize with modulation (https://github.com/nipy/nipype/pull/1023)
- ENH: Usability improvements in cluster environments (https://github.com/nipy/nipype/pull/1025)
- ENH: ANTs JointFusion() (https://github.com/nipy/nipype/pull/1042)
- ENH: Added csvReader() utility (https://github.com/nipy/nipype/pull/1044)
- FIX: typo in nipype.interfaces.freesurfer.utils.py Tkregister2 (https://github.com/nipy/nipype/pull/1083)
- FIX: SSHDataGrabber outputs now return full path to the grabbed/downloaded files. (https://github.com/nipy/nipype/pull/1086)
- FIX: Add QA output for TSNR to resting workflow (https://github.com/nipy/nipype/pull/1088)
- FIX: Change N4BiasFieldCorrection to use short tag for dimensionality (backward compatible) (https://github.com/nipy/nipype/pull/1096)
- ENH: Added -newgrid input to Warp in AFNI (3dWarp wrapper) (https://github.com/nipy/nipype/pull/1128)
- FIX: Fixed AFNI Copy interface to use positional inputs as required (https://github.com/nipy/nipype/pull/1131)
- ENH: Added a check in Dcm2nii to check if nipype created the config.ini file and remove if true (https://github.com/nipy/nipype/pull/1132)
- **ENH: Use a while loop to wait for Xvfb (up to a max wait time "xvfb_max_wait" in config file, default 10)** (https://github.com/nipy/nipype/pull/1142)

2.2 Release 0.10.0 (October 10, 2014)

- **ENH: New miscelaneous interfaces: SplitROIs (mapper), MergeROIs (reducer)** to enable parallel processing of very large images.

- **ENH: Updated FSL interfaces: BEDPOSTX and XFibres, former interfaces are still** available with the version suffix: BEDPOSTX4 and XFibres4. Added gpu versions of BEDPOSTX: BEDPOSTXGPU, BEDPOSTX5GPU, and BEDPOSTX4GPU
- ENH: Added experimental support for MIPAV algorithms thorugh JIST plugins
- ENH: New dipy interfaces: Denoise, Resample
- ENH: New Freesurfer interfaces: Tkregister2 (for conversion of fsl style matrices to freesurfer format), MRIPretess
- ENH: New FSL interfaces: WarpPoints, WarpPointsToStd, EpiReg, ProbTrackX2, WarpUtils, ConvertWarp
- ENH: New miscelaneous interfaces: AddCSVRow, NormalizeProbabilityMapSet, AddNoise
- ENH: New AFNI interfaces: Eval, Means, SVMTest, SVMTrain
- **ENH: FUGUE interface has been refactored to use the name_template system, 3 examples** added to doctests, some bugs solved.
- **API: Interfaces to external packages are no longer available in the top-level** `nipype` namespace, and must be imported directly (e.g. `from nipype.interfaces import fsl`).
- **ENH: Support for elastix via a set of new interfaces: Registration, ApplyWarp,** AnalyzeWarp, PointsWarp, and EditTransform
- ENH: New ANTs interface: ApplyTransformsToPoints, LaplacianThickness
- ENH: New Diffusion Toolkit interface: TrackMerge
- ENH: New MRtrix interface: FilterTracks
- **ENH: New metrics group in algorithms. Now Distance, Overlap, and FuzzyOverlap** are found in nipype.algorithms.metrics instead of misc. Overlap interface extended to allow files containing multiple ROIs and volume physical units.
- ENH: New interface in algorithms.metrics: ErrorMap (a voxel-wise diff map).
- ENH: New FreeSurfer workflow: create_skullstripped_recon_flow()
- **ENH: Deep revision of workflows for correction of dMRI artifacts. New dmri_preprocessing** example.
- ENH: New data grabbing interface that works over SSH connections, SSHDataGrabber
- ENH: New color mode for write_graph
- ENH: You can now force MapNodes to be run serially
- ENH: Added ANTS based openfmri workflow
- ENH: MapNode now supports flattening of nested lists
- ENH: Support for headless mode using Xvfb
- ENH: nipype_display_crash has a debugging mode
- FIX: MRTrix tracking algorithms were ignoring mask parameters.
- FIX: FNIRT registration pathway and associated OpenFMRI example script
- FIX: spm12b compatibility for Model estimate
- FIX: Batch scheduler controls the number of maximum jobs properly
- FIX: Update for FSL 5.0.7 which deprecated Contrast Manager

2.3 Release 0.9.2 (January 31, 2014)

- FIX: DataFinder was broken due to a typo
- FIX: Order of DataFinder outputs was not guaranteed, it's human sorted now
- ENH: New interfaces: Vnifti2Image, VtoMat

2.4 Release 0.9.1 (December 25, 2013)

- FIX: installation issues

2.5 Release 0.9.0 (December 20, 2013)

- ENH: SelectFiles: a streamlined version of DataGrabber
- ENH: new tools for defining workflows: JoinNode, synchronize and itersource

- ENH: W3C PROV support with optional RDF export built into Nipype
- ENH: Added support for Simple Linux Utility Resource Management (SLURM)
- **ENH: AFNI interfaces refactor, prefix, suffix are replaced by** "flexible_%s_templates"
- **ENH: New SPM interfaces:**
  spm.ResliceToReference,
  spm.DicomImport
- **ENH: New AFNI interfaces:**
  afni.AFNItoNIFTI
  afni.TCorr1D
- **ENH: Several new interfaces related to Camino were added:**
  camino.SFPICOCalibData
  camino.Conmat
  camino.QBallMX
  camino.LinRecon
  camino.SFPeaks
  One outdated interface no longer part of Camino was removed: - camino.Conmap
- **ENH: Three new mrtrix interfaces were added:**
  mrtrix.GenerateDirections
  mrtrix.FindShPeaks
  mrtrix.Directions2Amplitude
- **ENH: New FSL interfaces:**
  fsl.PrepareFieldmap
  fsl.TOPUP
  fsl.ApplyTOPUP
  fsl.Eddy
- **ENH: New misc interfaces:**
  FuzzyOverlap,
  P2PDistance
- ENH: New workflows: nipype.workflows.dmri.fsl.epi.[fieldmap_correction&topup_correction]
- ENH: Added simplified outputname generation for command line interfaces.
- ENH: Allow ants use a single mask image
- ENH: Create configuration option for parameterizing directories with hashes
- ENH: arrange nodes by topological sort with disconnected subgraphs
- ENH: uses the nidm iri namespace for uuids
- ENH: remove old reporting webpage
- ENH: Added support for Vagrant
- API: 'name' is now a positional argument for Workflow, Node, and MapNode constructors
- API: SPM now defaults to SPM8 or SPM12b job format
- API: DataGrabber and SelectFiles use human (or natural) sort now
- **FIX: Several fixes related to Camino interfaces:**
  ProcStreamlines would ignore many arguments silently (target, waypoint, exclusion ROIS, etc.)
  DTLUTGen would silently round the "step", "snr" and "trace" parameters to integers
  PicoPDFs would not accept more than one lookup table
  PicoPDFs default pdf did not correspond to Camino default
  Track input model names were outdated (and would generate an error)
  Track numpds parameter could not be set for deterministic tractography
  FA created output files with erroneous extension
- FIX: Deals properly with 3d files in SPM Realign
- FIX: SPM with MCR fixed
- FIX: Cleaned up input and output spec metadata
- FIX: example openfmri script now makes the contrast spec a hashed input
- FIX: FILMGLS compatibility with FSL 5.0.5
- FIX: Freesurfer recon-all resume now avoids setting inputs
- FIX: File removal from node respects file associations img/hdr/mat, BRIK/HEAD

2.6 Release 0.8.0 (May 8, 2013)

- **ENH: New interfaces: nipy.Trim, fsl.GLM, fsl.SigLoss, spm.VBMSegment, fsl.InvWarp,** dipy.TensorMode
- ENH: Allow control over terminal output for commandline interfaces
- ENH: Added preliminary support for generating Python code from Workflows.
- **ENH: New workflows for dMRI and fMRI pre-processing: added motion artifact correction** with rotation of the B-matrix, and susceptibility correction for EPI imaging using fieldmaps. Updated eddy_correct pipeline to support both dMRI and fMRI, and new parameters.
- ENH: Minor improvements to FSL's FUGUE and FLIRT interfaces
- ENH: Added optional dilation of parcels in cmtk.Parcellate
- ENH: Interpolation mode added to afni.Resample
- **ENH: Function interface can accept a list of strings containing import statements** that allow external functions to run without their imports defined in the function body
- ENH: Allow node configurations to override master configuration
- FIX: SpecifyModel works with 3D files correctly now.

2.7 Release 0.7.0 (Dec 18, 2012)

- ENH: Add basic support for LSF plugin.
- **ENH: New interfaces: ICC, Meshfix, ants.Register, C3dAffineTool, ants.JacobianDeterminant,** afni.AutoTcorrelate, DcmStack
- ENH: New workflows: ants template building (both using 'ANTS' and the new 'antsRegistration')
- **ENH: New examples: how to use ANTS template building workflows (smri_ants_build_tmeplate),** how to set SGE specific options (smri_ants_build_template_new)
- ENH: added no_flatten option to Merge
- ENH: added versioning option and checking to traits
- ENH: added deprecation metadata to traits
- ENH: Slicer interfaces were updated to version 4.1

2.8 Release 0.6.0 (Jun 30, 2012)

- API: display variable no longer encoded as inputs in commandline interfaces
- ENH: input hash not modified when environment DISPLAY is changed
- ENH: support for 3d files for TSNR calculation
- ENH: Preliminary support for graph submission with SGE, PBS and Soma Workflow
- **ENH: New interfaces: MySQLSink, nipy.Similarity, WatershedBEM, MRIsSmooth,** NetworkBasedStatistic, Atropos, N4BiasFieldCorrection, ApplyTransforms, fs.MakeAverageSubject, epidewarp.fsl, WarpTimeSeriesImageMultiTransform, AVScale, mri_ms_LDA
- ENH: simple interfaces for spm
- FIX: CompCor component calculation was erroneous
- FIX: filename generation for AFNI and PRELUDE
- FIX: improved slicer module autogeneration
- FIX: added missing options for BBRegsiter
- FIX: functionality of remove_unnecessary_ouputs cleaned up
- FIX: local hash check works with appropriate inputs
- FIX: Captures all stdout from commandline programs
- FIX: Afni outputs should inherit from TraitedSpec

2.9 Release 0.5.3 (Mar 23, 2012)

- FIX: SPM model generation when output units is in scans

2.10 Release 0.5.2 (Mar 14, 2012)

- API: Node now allows specifying node level configuration for SGE/PBS clusters
- API: Logging to file is disabled by default
- API: New location of log file -> .nipype/nipype.cfg
- ENH: Changing logging options via config works for distributed processing
- FIX: Unittests on debian (logging and ipython)

2.11 Release 0.5 (Mar 10, 2012)

- API: FSL defaults to Nifti when OUTPUTTYPE environment variable not found
- API: By default inputs are removed from Node working directory
- API: InterfaceResult class is now versioned and stores class type not instance
- API: Added FIRST interface
- **API: Added max_jobs paramter to plugin_args. limits the number of jobs** executing at any given point in time
- API: crashdump_dir is now a config execution option
- **API: new config execution options for controlling hash checking, execution and** logging behavior when running in distributed mode.
- API: Node/MapNode has new attribute that allows it to run on master thread.
- API: IPython plugin now invokes IPython 0.11 or greater
- API: Canned workflows are now all under a different package structure
- API: SpecifyModel event_info renamed to event_files
- **API: DataGrabber is always being rerun (unless overwrite is set to False on** Node level)
- **API: "stop_on_first_rerun" does not stop for DataGrabber (unless overwrite is** set to True on Node level)
- **API: Output prefix can be set for spm nodes (SliceTiming, Realign, Coregister,** Normalize, Smooth)
- ENH: Added fsl resting state workflow based on behzadi 2007 CompCorr method.
- ENH: TSNR node produces mean and std-dev maps; allows polynomial detrending
- ENH: IdentityNodes are removed prior to execution
- ENH: Added Michael Notter's beginner's guide
- ENH: Added engine support for status callback functions
- ENH: SPM create warped node
- ENH: All underlying interfaces (including python ones) are now optional
- ENH: Added imperative programming option with Nodes and caching
- ENH: Added debug mode to configuration
- ENH: Results can be stored and loaded without traits exceptions
- ENH: Added concurrent log handler for distributed writing to log file
- ENH: Reporting can be turned off using config
- ENH: Added stats files to FreeSurferOutput
- ENH: Support for Condor through qsub emulation
- **ENH: IdentityNode with iterable expansion takes place after remaining Identity** Node removal
- ENH: Crashfile display script added
- ENH: Added FmriRealign4d node wrapped from nipy
- ENH: Added TBSS workflows and examples
- ENH: Support for openfmri data processing
- ENH: Package version check
- FIX: Fixed spm preproc workflow to cater to multiple functional runs
- FIX: Workflow outputs displays nodes with empty outputs
- FIX: SUSAN workflow works without usans
- FIX: SGE fixed for reading custom templates
- FIX: warping in SPM realign, Dartel and interpolation parameters
- FIX: Fixed voxel size parameter in freesurfer mri_convert
- FIX: 4D images in spm coregister

- FIX: Works around matlab tty bug
- FIX: Overwriting connection raises exception
- **FIX: Outputs are loaded from results and not stored in memory for during** distributed operation
- FIX: SPM threshold uses SPM.mat name and improved error detection
- FIX: Removing directory contents works even when a node has no outputs
- FIX: DARTEL workflows will run only when SPM 8 is available
- FIX: SPM Normalize estimate field fixed
- FIX: hashmethod argument now used for calculating hash of old file
- FIX: Modelgen now allows FSL style event files

2.12 Release 0.4.1 (Jun 16, 2011)

- Minor bugfixes

2.13 Release 0.4 (Jun 11, 2011)

- **API: Timestamp hashing does not use ctime anymore. Please update your hashes by** running workflows with updatehash=True option NOTE: THIS IS THE DEFAULT CONFIG NOW, so unless you updatehash, workflows will rerun
- **API: Workflow run function no longer supports (inseries, createdirsonly).** Functions used in connect string must be pickleable
- API: SPM EstimateContrast: ignore_derivs replaced by use_derivs
- API: All interfaces: added new config option ignore_exception
- **API: SpecifModel no longer supports (concatenate_runs, output_specs). high_pass_filter** cutoff is mandatory (even if its set to np.inf). Additional interfaces SpecifySPMModel and SpecifySparseModel support other types of data.
- API: fsl.DTIFit input "save" is now called "save_tensor"
- **API: All inputs of IdentityInterfaces are mandatory by default. You can turn** this off by specifying mandatory_inputs=False to the constructor.
- API: fsl FILMGLS input "autocorr_estimate" is now called "autocorr_estimate_only"
- **API: fsl ContrastMgr now requires access to specific files (no longer accepts** the result directory)
- **API: freesurfer.GLMFit input "surf" is now a boolean with three corresponding** inputs – subject_id, hemi, and surf_geo
- ENH: All commandline interfaces display stdout and stderr
- ENH: All interfaces raise exceptions on error with an option to suppress
- **ENH: Supports a plugin interface for execution (current support for multiprocessing,** IPython, SGE, PBS)
- ENH: MapNode runs in parallel under IPython, SGE, MultiProc, PBS
- ENH: Optionally allows keeping only required outputs
- **ENH: New interface: utility.Rename to change the name of files, optionally** using python string-formatting with inputs or regular expressions matching
- ENH: New interface: freesurfer.ApplyMask (mri_mask)
- ENH: New FSL interface – SwapDimensions (fslswapdim)
- ENH: Sparse models allow regressor scaling and temporal derivatives
- **ENH: Added support for the component parts of FSL's TBSS workflow (TBSSSkeleton** and DistanceMap)
- ENH: dcm2nii interface exposes bvals, bvecs, reoriented and cropped images
- **ENH: Added several higher-level interfaces to the fslmaths command:** ChangeDataType, Threshold, MeanImage, IsotropicSmooth, ApplyMask, TemporalFilter DilateImage, ErodeImage, SpatialFilter, UnaryMaths, BinaryMaths, MultiImageMaths
- ENH: added support for networx 1.4 and improved iterable expansion
- ENH: Replaced BEDPOSTX and EddyCurrent with nipype pipelines
- ENH: Ability to create a hierarchical dot file

- ENH: Improved debugging information for rerunning nodes
- ENH: Added 'stop_on_first_rerun' option
- ENH: Added support for Camino
- ENH: Added support for Camino2Trackvis
- ENH: Added support for Connectome Viewer
- BF: dcm2nii interface handles gzipped files correctly
- BF: FNIRT generates proper outputs
- BF: fsl.DTIFit now properly collects tensor volume
- BF: updatehash now removes old result hash file

2.14 Release 0.3.4 (Jan 12, 2011)

- API: hash values for float use a string conversion up to the 10th decimal place.
- API: Iterables in output path will always be generated as _var1_val1_var2_val2 pairs
- ENH: Added support to nipy: GLM fit, contrast estimation and calculating mask from EPI
- **ENH: Added support for manipulating surface files in Freesurfer:**
  projecting volume images onto the surface
  smoothing along the surface
  transforming a surface image from one subject to another
  using tksurfer to save pictures of the surface
- ENH: Added support for flash processing using FreeSurfer
- ENH: Added support for flirt matrix in BBRegister
- ENH: Added support for FSL convert_xfm
- ENH: hashes can be updated again without rerunning all nodes.
- ENH: Added multiple regression design for FSL
- ENH: Added SPM based Analyze to Nifti converter
- ENH: Added increased support for PyXNAT
- ENH: Added support for MCR-based binary version of SPM
- ENH: Added SPM node for calculating various threshold statistics
- ENH: Added distance and dissimilarity measurements
- BF: Diffusion toolkit gets installed
- **BF: Changed FNIRT interface to accept flexible lists (rather than 4-tuples)** on all options specific to different subsampling levels

2.15 Release 0.3.3 (Sep 16, 2010)

- API: subject_id in ModelSpec is now deprecated
- API: spm.Threshold - does not need mask, beta, RPV anymore - takes only one image (stat_image - mind the name change) - works with SPM2 SPM.mat - returns additional map - pre topological FDR
- ENH: Added support for Diffusion toolkit
- ENH: Added support for FSL slicer and overlay
- ENH: Added support for dcm2nii
- BF: DataSink properly handles lists of lists now
- BF: DataGrabber has option for raising Exception on getting empty lists
- BF: Traits logic for 'requires' metadata
- BF: allows workflows to be relocatable
- BF: nested workflows with connections don't raise connection not found error
- BF: multiple workflows with identical nodenames and iterables do not create nestsed workflows

2.16 Release 0.3.2 (Aug 03, 2010)

2.16.1 Enhancements

- all outputs from nodes are now pickled as part of workflow processing
- added git developer docs

2.16.2 Bugs fixed

- FreeSurfer
- Fixed bugs in SegStats doctest

2.17 Release 0.3.1 (Jul 29, 2010)

2.17.1 Bugs fixed

- FreeSurfer
- Fixed bugs in glmfit and concatenate
- Added group t-test to freesurfer tutorial

2.18 Release 0.3 (Jul 27, 2010)

2.18.1 Incompatible changes

- Complete redesign of the Interface class - heavy use of Traits.
- Changes in the engine API - added Workflow and MapNode. Compulsory name argument.

2.18.2 Features added

- General:
- Type checking of inputs and outputs using Traits from ETS.
- Support for nested workflows.
- Preliminary Slicer and AFNI support.
- New flexible DataGrabber node.
- AtlasPick and Threshold nodes.
- Preliminary support for XNAT.
- Doubled number of the tutorials.
- FSL:
- Added DTI processing nodes (note that TBSS nodes are still experimental).
- Recreated FEAT workflow.
- SPM:
- Added New Segment and many other nodes.
- Redesigned second level analysis.
- Developer

CHAPTER 3

API

Release 0.11.0
Date September 15, 2015, 17:26 PDT

CHAPTER 4

Developer Guide

Release 0.11.0
Date September 15, 2015, 17:26 PDT

Since nipype is part of the NIPY project, we follow the same conventions documented in the NIPY Developers Guide. For bleeding-edge version help see Nightly documentation

4.1 Interface Specifications

4.1.1 Before you start

Nipype is a young project maintained by an enthusiastic group of developers. Even though the documentation might be sparse or cryptic at times we strongly encourage you to contact us on the official nipype developers mailing list in case of any troubles: nipy-devel@neuroimaging.scipy.org (we are sharing a mailing list with the nipy community therefore please add `[nipype]` to the messsage title).

4.1.2 Overview

We're using the Enthought Traits package for all of our inputs and outputs. Traits allows us to validate user inputs and provides a mechanism to handle all the *special cases* in a simple and concise way though metadata. With the metadata, each input/output can have an optional set of metadata attributes (described in more detail below). The machinery for handling the metadata is located in the base classes, so all subclasses use the same code to handle these cases. This is in contrast to our previous code where every class defined it's own _parse_inputs, run and aggregate_outputs methods to handle these cases. Which of course leads to a dozen different ways to solve the same problem.

Traits is a big package with a lot to learn in order to take full advantage of. But don't be intimidated! To write a Nipype Trait Specification, you only need to learn a few of the basics of Traits. Here are a few starting points in the documentation:

- What are Traits? The Introduction in the User Manual gives a brief description of the functionality traits provides.
- Traits and metadata. The second section of the User Manual gives more details on traits and how to use them. Plus there a section describing metadata, including the metadata all traits have.
- If your interested in more of a *big picture* overview, Gael wrote a good tutorial that shows how to write a scientific application using traits for the benefit of the generated UI components. (For now, Nipype is not taking advantage of the generated UI feature of traits.)

Traits version

We're using Traits version 3.x which can be install as part of EPD or from pypi

More documentation

Not everything is documented in the User Manual, in those cases the enthought-dev mailing list or the API docs is your next place to look.

4.1.3 Nipype Interface Specifications

Each interface class defines two specifications: 1) an InputSpec and 2) an OutputSpec. Each of these are prefixed with the class name of the interfaces. For example, Bet has these specs:

- BETInputSpec
- BETOutputSpec

Each of these Specs are classes, derived from a base TraitedSpec class (more on these below). The InputSpec consists of attributes which correspond to different parameters for the tool they wrap/interface. In the case of a command-line tool like Bet, the InputSpec attributes correspond to the different command-line parameters that can be passed to Bet. If you are familiar with the Nipype 0.2 code-base, these attributes are the same as the keys in the opt_map dictionaries. When an interfaces class is instantiated, the InputSpec is bound to the `inputs` attribute of that object. Below is an example of how the `inputs` appear to a user for Bet:

```
>>> from nipype.interfaces import fsl
>>> bet = fsl.BET()
>>> type(bet.inputs)
<class 'nipype.interfaces.fsl.preprocess.BETInputSpec'>
>>> bet.inputs.<TAB>
bet.inputs.__class__              bet.inputs.center
bet.inputs.__delattr__            bet.inputs.environ
bet.inputs.__doc__                bet.inputs.frac
bet.inputs.__getattribute__       bet.inputs.functional
bet.inputs.__hash__               bet.inputs.hashval
bet.inputs.__init__               bet.inputs.infile
bet.inputs.__new__                bet.inputs.items
bet.inputs.__reduce__             bet.inputs.mask
bet.inputs.__reduce_ex__          bet.inputs.mesh
bet.inputs.__repr__               bet.inputs.nooutput
bet.inputs.__setattr__            bet.inputs.outfile
bet.inputs.__str__                bet.inputs.outline
bet.inputs._generate_handlers     bet.inputs.outputtype
bet.inputs._get_hashval           bet.inputs.radius
bet.inputs._hash_infile           bet.inputs.reduce_bias
bet.inputs._xor_inputs            bet.inputs.skull
bet.inputs._xor_warn              bet.inputs.threshold
bet.inputs.args                   bet.inputs.vertical_gradient
```

Each Spec inherits from a parent Spec. The parent Specs provide attribute(s) that are common to all child classes. For example, FSL InputSpecs inherit from interfaces.fsl.base.FSLTraitedSpec. FSLTraitedSpec defines an `outputtype` attribute, which stores the file type (NIFTI, NIFTI_PAIR, etc...) for all generated output files.

InputSpec class hierarchy

Below is the current class hierarchy for InputSpec classes (from base class down to subclasses).:

`TraitedSpec`: Nipype's primary base class for all Specs. Provides initialization, some nipype-specific methods and any trait handlers we define. Inherits from traits.HasTraits.

`BaseInterfaceInputSpec`: Defines inputs common to all Interfaces (`ignore_exception`). If in doubt inherit from this.

`CommandLineInputSpec`: Defines inputs common to all command-line classes (`args` and `environ`)

`FSLTraitedSpec`: Defines inputs common to all FSL classes (`outputtype`)

`SPMCommandInputSpec`: Defines inputs common to all SPM classes (`matlab_cmd`, `path`, and `mfile`)

`FSTraitedSpec`: Defines inputs common to all FreeSurfer classes (`sbjects_dir`)

`MatlabInputSpec`: Defines inputs common to all Matlab classes (`script`, `nodesktop`, `nosplash`, `logfile`,

single_comp_thread, mfile, script_file, and paths)
SlicerCommandLineInputSpec: Defines inputs common to all Slicer classes (module)

Most developers will only need to code at the the interface-level (i.e. implementing custom class inheriting from one of the above classes).

Output Specs

The OutputSpec defines the outputs that are generated, or possibly generated depending on inputs, by the tool. OutputSpecs inherit from `interfaces.base.TraitedSpec` directly.

4.1.4 Traited Attributes

Each specification attribute is an instance of a Trait class. These classes encapsulate many standard Python types like Float and Int, but with additional behavior like type checking. (*See the documentation on traits for more information on these trait types.*) To handle unique behaviors of our attributes we us traits metadata. These are keyword arguments supplied in the initialization of the attributes. The base classes `BaseInterface` and `CommandLine` (defined in `nipype.interfaces.base`) check for the existence/or value of these metadata and handle the inputs/outputs accordingly. For example, all mandatory parameters will have the `mandatory = True` metadata:

```
class BetInputSpec(FSLTraitedSpec):
  infile = File(exists=True,
                desc = 'input file to skull strip',
                argstr='%s', position=0, mandatory=True)
```

Common

exists For files, use `nipype.interfaces.base.File` as the trait type. If the file must exist for the tool to execute, specify `exists = True` in the initialization of File (as shown in BetInputSpec above). This will trigger the underlying traits code to confirm the file assigned to that *input* actually exists. If it does not exist, the user will be presented with an error message:

```
>>> bet.inputs.infile = 'does_not_exist.nii'
------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython console>", line 1, in <module>
  File "/Users/cburns/local/lib/python2.5/site-packages/nipype/interfaces/base.py", line 76,
    self.error( object, name, value )
  File "/Users/cburns/local/lib/python2.5/site-packages/enthought/traits/trait_handlers.py",
    value )
TraitError: The 'infile' trait of a BetInputSpec instance must be a file
name, but a value of 'does_not_exist.nii' <type 'str'> was specified.
```

hash_files To be used with inputs that are defining output filenames. When this flag is set to false any Nipype will not try to hash any files described by this input. This is useful to avoid rerunning when the specified output file already exists and has changed.

desc All trait objects have a set of default metadata attributes. `desc` is one of those and is used as a simple, one-line docstring. The `desc` is printed when users use the `help()` methods.
Required: This metadata is required by all nipype interface classes.

usedefault Set this metadata to True when the *default value* for the trait type of this attribute is an acceptable value. All trait objects have a default value, `traits.Int` has a default of `0`, `traits.Float` has a default of `0.0`, etc... You can also define a default value when you define the class. For example, in the code below all objects of `Foo` will have a default value of 12 for `x`:

```
>>> import enthought.traits.api as traits
>>> class Foo(traits.HasTraits):
```

```
...     x = traits.Int(12)
...     y = traits.Int
...
>>> foo = Foo()
>>> foo.x
12
>>> foo.y
0
```

Nipype only passes `inputs` on to the underlying package if they have been defined (more on this later). So if you specify `usedefault = True`, you are telling the parser to pass the default value on to the underlying package. Let's look at the InputSpec for SPM Realign:

```
class RealignInputSpec(BaseInterfaceInputSpec):
    jobtype = traits.Enum('estwrite', 'estimate', 'write',
                          desc='one of: estimate, write, estwrite',
                          usedefault=True)
```

Here we've defined `jobtype` to be an enumerated trait type, `Enum`, which can be set to one of the following: `estwrite`, `estimate`, or `write`. In a container, the default is always the first element. So in this case, the default will be `estwrite`:

```
>>> from nipype.interfaces import spm
>>> rlgn = spm.Realign()
>>> rlgn.inputs.infile
<undefined>
>>> rlgn.inputs.jobtype
'estwrite'
```

`xor` and `requires` Both of these accept a list of trait names. The `xor` metadata reflects mutually exclusive traits, while the requires metadata reflects traits that have to be set together. When a xor-ed trait is set, all other traits belonging to the list are set to Undefined. The function check_mandatory_inputs ensures that all requirements (both mandatory and via the requires metadata are satisfied). These are also reflected in the help function.

`copyfile` This is metadata for a File or Directory trait that is relevant only in the context of wrapping an interface in a *Node* and *MapNode*. *copyfile* can be set to either *True* or *False*. *False* indicates that contents should be symlinked, while *True* indicates that the contents should be copied over.

`min_ver` and `max_ver` These metadata determine if a particular trait will be available when a given version of the underlying interface runs. Note that this check is performed at runtime.:

```
class RealignInputSpec(BaseInterfaceInputSpec):
    jobtype = traits.Enum('estwrite', 'estimate', 'write', min_ver='5',
                          usedefault=True)
```

`deprecated` and `new_name` This is metadata for removing or renaming an input field from a spec.:

```
class RealignInputSpec(BaseInterfaceInputSpec):
    jobtype = traits.Enum('estwrite', 'estimate', 'write',
                          deprecated='0.8',
                          desc='one of: estimate, write, estwrite',
                          usedefault=True)
```

In the above example this means that the *jobtype* input is deprecated and will be removed in version 0.8. Deprecation should be set to two versions from current release. Raises *TraitError* after package version crosses the deprecation version.
For inputs that are being renamed, one can specify the new name of the field.:

```
class RealignInputSpec(BaseInterfaceInputSpec):
    jobtype = traits.Enum('estwrite', 'estimate', 'write',
                          deprecated='0.8', new_name='job_type',
                          desc='one of: estimate, write, estwrite',
```

```
                                usedefault=True)
        job_type = traits.Enum('estwrite', 'estimate', 'write',
                               desc='one of: estimate, write, estwrite',
                               usedefault=True)
```

In the above example, the *jobtype* field is being renamed to *job_type*. When *new_name* is provided it must exist as a trait, otherwise an exception will be raised.

Note: The version information for *min_ver*, *max_ver* and *deprecated* has to be provided as a string. For example, *min_ver='0.1'*.

CommandLine

argstr The metadata keyword for specifying the format strings for the parameters. This was the *value* string in the opt_map dictionaries of Nipype 0.2 code. If we look at the `FlirtInputSpec`, the `argstr` for the reference file corresponds to the argument string I would need to provide with the command-line version of `flirt`:

```
    class FlirtInputSpec(FSLTraitedSpec):
        reference = File(exists = True, argstr = '-ref %s', mandatory = True,
                         position = 1, desc = 'reference file')
```

Required: This metadata is required by all command-line interface classes.

position This metadata is used to specify the position of arguments. Both positive and negative values are accepted. `position = 0` will position this argument as the first parameter after the command name. `position = -1` will position this argument as the last parameter, after all other parameters.

genfile If True, the `genfile` metadata specifies that a filename should be generated for this parameter *if-and-only-if* the user did not provide one. The nipype convention is to automatically generate output filenames when not specified by the user both as a convenience for the user and so the pipeline can easily gather the outputs. Requires `_gen_filename()` method to be implemented. This way should be used if the desired file name is dependent on some runtime variables (such as file name of one of the inputs, or current working directory). In case when it should be fixed it's recommended to just use `usedefault`.

sep For List traits the string with witch elements of the list will be joined.

name_source Indicates the list of input fields from which the value of the current File output variable will be drawn. This input field must be the name of a File. Chaining is allowed, meaning that an input field can point to another as `name_source`, which also points as `name_source` to a third field. In this situation, the templates for substitutions are also accumulated.

name_template By default a `%s_generated` template is used to create the output filename. This metadata keyword allows overriding the generated name.

keep_extension Use this and set it `True` if you want the extension from the input to be kept.

SPM

field name of the structure refered by the SPM job manager

Required: This metadata is required by all SPM-mediated interface classes.

4.1.5 Defining an interface class

Common

When you define an interface class, you will define these attributes and methods:

- `input_spec`: the InputSpec
- `output_spec`: the OutputSpec
- `_list_outputs()`: Returns a dictionary containing names of generated files that are expected after package completes execution. This is used by `BaseInterface.aggregate_outputs` to gather all output files for the pipeline.

CommandLine

For command-line interfaces:

- `_cmd`: the command-line command
  If you used genfile:
- `_gen_filename(name)`: Generate filename, used for filenames that nipype generates as a convenience for users. This is for parameters that are required by the wrapped package, but we're generating from some other parameter. For example, `BET.inputs.outfile` is required by BET but we can generate the name from `BET.inputs.infile`. Override this method in subclass to handle.
  And optionally:
- `_redirect_x`: If set to True it will make Nipype start Xvfb before running the interface and redirect X output to it. This is useful for
  commandlines that spawn a graphical user interface.
- `_format_arg(name, spec, value)`: For extra formatting of the input values before passing them to generic `_parse_inputs()` method.
  For example this is the class definition for Flirt, minus the docstring:

```
    class FLIRTInputSpec(FSLCommandInputSpec):
        in_file = File(exists=True, argstr='-in %s', mandatory=True,
                       position=0, desc='input file')
        reference = File(exists=True, argstr='-ref %s', mandatory=True,
                         position=1, desc='reference file')
        out_file = File(argstr='-out %s', desc='registered output file',
                        name_source=['in_file'], name_template='%s_flirt',
                        position=2, hash_files=False)
        out_matrix_file = File(argstr='-omat %s',
                               name_source=['in_file'], keep_extension=True,
                               name_template='%s_flirt.mat',
                               desc='output affine matrix in 4x4 asciii format',
                               position=3, hash_files=False)
        out_log = File(name_source=['in_file'], keep_extension=True,
                       requires=['save_log'],
                       name_template='%s_flirt.log', desc='output log')
        ...

    class FLIRTOutputSpec(TraitedSpec):
        out_file = File(exists=True,
                        desc='path/name of registered file (if generated)')
        out_matrix_file = File(exists=True,
                               desc='path/name of calculated affine transform '
                               '(if generated)')
        out_log = File(desc='path/name of output log (if generated)')

    class Flirt(FSLCommand):
        _cmd = 'flirt'
        input_spec = FlirtInputSpec
        output_spec = FlirtOutputSpec
```

There are two possible output files `outfile` and `outmatrix`, both of which can be generated if not specified by the user.
Also notice the use of `self._gen_fname()` - a FSLCommand helper method for generating filenames (with extensions conforming with FSLOUTPUTTYPE).
See also How to wrap a command line tool.

SPM

For SPM-mediated interfaces:

- `_jobtype` and `_jobname`: special names used used by the SPM job manager. You can find them by saving

your batch job as an .m file and looking up the code.
And optionally:

- `_format_arg(name, spec, value)`: For extra formatting of the input values before passing them to generic `_parse_inputs()` method.

Matlab

See How to wrap a MATLAB script.

Python

See How to wrap a Python script.

4.1.6 Undefined inputs

All the inputs and outputs that were not explicitly set (And do not have a usedefault flag - see above) will have Undefined value. To check if something is defined you have to explicitly call `isdefiend` function (comparing to None will not work).

4.1.7 Example of inputs

Below we have an example of using Bet. We can see from the help which inputs are mandatory and which are optional, along with the one-line description provided by the `desc` metadata:

```
>>> from nipype.interfaces import fsl
>>> fsl.BET.help()
Inputs
------

Mandatory:
 infile: input file to skull strip

Optional:
 args: Additional parameters to the command
 center: center of gravity in voxels
 environ: Environment variables (default={})
 frac: fractional intensity threshold
 functional: apply to 4D fMRI data
 mask: create binary mask image
 mesh: generate a vtk mesh brain surface
 nooutput: Don't generate segmented output
 outfile: name of output skull stripped image
 outline: create surface outline image
 outputtype: None
 radius: head radius
 reduce_bias: bias field and neck cleanup
 skull: create skull image
 threshold: apply thresholding to segmented brain image and mask
 vertical_gradient: vertical gradient in fractional intensity threshold (-1, 1)

Outputs
-------
maskfile: path/name of binary brain mask (if generated)
meshfile: path/name of vtk mesh file (if generated)
outfile: path/name of skullstripped file
outlinefile: path/name of outline file (if generated)
```

Here we create a bet object and specify the required input. We then check our inputs to see which are defined and which are not:

```
>>> bet = fsl.BET(infile = 'f3.nii')
>>> bet.inputs
args = <undefined>
center = <undefined>
environ = {'FSLOUTPUTTYPE': 'NIFTI_GZ'}
frac = <undefined>
functional = <undefined>
infile = f3.nii
mask = <undefined>
mesh = <undefined>
nooutput = <undefined>
outfile = <undefined>
outline = <undefined>
outputtype = NIFTI_GZ
radius = <undefined>
reduce_bias = <undefined>
skull = <undefined>
threshold = <undefined>
vertical_gradient = <undefined>
>>> bet.cmdline
'bet f3.nii /Users/cburns/data/nipype/s1/f3_brain.nii.gz'
```

We also checked the command-line that will be generated when we run the command and can see the generated output filename `f3_brain.nii.gz`.

4.2 How to wrap a command line tool

The aim of this section is to describe how external programs and scripts can be wrapped for use in Nipype either as interactive interfaces or within the workflow/pipeline environment. Currently, there is support for command line executables/scripts and matlab scripts. One can also create pure Python interfaces. The key to defining interfaces is to provide a formal specification of inputs and outputs and determining what outputs are generated given a set of inputs.

4.2.1 Defining inputs and outputs

In Nipype we use Enthought Traits to define inputs and outputs of the interfaces. This allows to introduce easy type checking. Inputs and outputs are grouped into separate classes (usually suffixed with InputSpec and OutputSpec). For example:

```
class ExampleInputSpec(TraitedSpec):
        input_volume = File(desc = "Input volume", exists = True,
                            mandatory = True)
        parameter = traits.Int(desc = "some parameter")

class ExampleOutputSpec(TraitedSpec):
        output_volume = File(desc = "Output volume", exists = True)
```

For the Traits (and Nipype) to work correctly output and input spec has to be inherited from TraitedSpec (however, this does not have to be direct inheritance).
Traits (File, Int etc.) have different parameters (called metadata). In the above example we have used the `desc` metadata which holds human readable description of the input. The `mandatory` flag forces Nipype to throw an exception if the input was not set. `exists` is a special flag that works only for `File traits` and checks if the provided file exists. More details can be found at Interface Specifications.
The input and output specifications have to be connected to the our example interface class:

```
class Example(Interface):
        input_spec = ExampleInputSpec
        output_spec = ExampleOutputSpec
```

Where the names of the classes grouping inputs and outputs were arbitrary the names of the fields within the interface they are assigned are not (it always has to be input_spec and output_spec). Of course this interface does not do much because we have not specified how to process the inputs and create the outputs. This can be done in many ways.

4.2.2 Command line executable

As with all interfaces command line wrappers need to have inputs defined. Command line input spec has to inherit from CommandLineInputSpec which adds two extra inputs: environ (a dictionary of environmental variables), and args (a string defining extra flags). In addition input spec can define the relation between the inputs and the generated command line. To achieve this we have added two metadata: `argstr` (string defining how the argument should be formated) and `position` (number defining the order of the arguments). For example

```
class ExampleInputSpec(CommandLineSpec):
        input_volume = File(desc = "Input volume", exists = True,
                            mandatory = True, position = 0, argstr="%s")
        parameter = traits.Int(desc = "some parameter", argstr = "--param %d")
```

As you probably noticed the `argstr` is a printf type string with formatting symbols. For an input defined in InputSpec to be included into the executed commandline `argstr` has to be included. Additionally inside the main interface class you need to specify the name of the executable by assigning it to the `_cmd` field. Also the main interface class needs to inherit from `nipype.interfaces.base.CommandLine`:

```
class Example(CommandLine):
        _cmd = 'my_command'
        input_spec = ExampleInputSpec
        output_spec = ExampleOutputSpec
```

There is one more thing we need to take care of. When the executable finishes processing it will presumably create some output files. We need to know which files to look for, check if they exist and expose them to whatever node would like to use them. This is done by implementing `_list_outputs` method in the main interface class. Basically what it does is assigning the expected output files to the fields of our output spec:

```
def _list_outputs(self):
        outputs = self.output_spec().get()
        outputs['output_volume'] = os.path.abspath('name_of_the_file_this_cmd_made.nii')
        return outputs
```

Sometimes the inputs need extra parsing before turning into command line parameters. For example imagine a parameter selecting between three methods: “old”, “standard” and “new”. Imagine also that the command line accept this as a parameter “–method=” accepting 0, 1 or 2. Since we are aiming to make nipype scripts as informative as possible it’s better to define the inputs as following:

```
class ExampleInputSpec(CommandLineSpec):
        method = traits.Enum("old", "standard", "new", desc = "method",
                             argstr="--method=%d")
```

Here we’ve used the Enum trait which restricts input a few fixed options. If we would leave it as it is it would not work since the argstr is expecting numbers. We need to do additional parsing by overloading the following method in the main interface class:

```
def _format_arg(self, name, spec, value):
        if name == 'method':
            return spec.argstr%{"old":0, "standard":1, "new":2}[value]
        return super(Example, self)._format_arg(name, spec, value)
```

Here is a minimalistic interface for the gzip command:

```
    from nipype.interfaces.base import (
        TraitedSpec,
        CommandLineInputSpec,
        CommandLine,
        File
    )
    import os

    class GZipInputSpec(CommandLineInputSpec):
        input_file = File(desc="File", exists=True, mandatory=True, argstr="%s")

    class GZipOutputSpec(TraitedSpec):
        output_file = File(desc = "Zip file", exists = True)

    class GZipTask(CommandLine):
        input_spec = GZipInputSpec
        output_spec = GZipOutputSpec
        cmd = 'gzip'

        def _list_outputs(self):
                outputs = self.output_spec().get()
                outputs['output_file'] = os.path.abspath(self.inputs.input_file + ".gz")
                return outputs

    if __name__ == '__main__':

        zipper = GZipTask(input_file='an_existing_file')
        print zipper.cmdline
        zipper.run()
```

4.2.3 Creating outputs on the fly

In many cases, command line executables will require specifying output file names as arguments on the command line. We have simplified this procedure with three additional metadata terms: `name_source`, `name_template`, `keep_extension`.
For example in the InvWarp class, the `inverse_warp` parameter is the name of the output file that is created by the routine.

```
    class InvWarpInputSpec(FSLCommandInputSpec):
        ...
        inverse_warp = File(argstr='--out=%s', name_source=['warp'],
                            hash_files=False, name_template='%s_inverse',
        ...
```

we add several metadata to inputspec.
name_source indicates which field to draw from, this field must be the name of a File.
hash_files indicates that the input for this field if provided should not be used in computing the input hash for this interface.
name_template (optional) overrides the default `_generated` suffix
output_name (optional) name of the output (if this is not set same name as the input will be assumed)
keep_extension (optional - not used) if you want the extension from the input to be kept
In addition one can add functionality to your class or base class, to allow changing extensions specific to package or interface

```
    def self._overload_extension(self, value):
        return value #do whatever you want here with the name
```

Finally, in the outputspec make sure the name matches that of the inputspec.

```
    class InvWarpOutputSpec(TraitedSpec):
        inverse_warp = File(exists=True,
                            desc=('Name of output file, containing warps that '
                            'are the "reverse" of those in --warp.'))
```

4.3 How to wrap a MATLAB script

This is minimal script for wrapping MATLAB code. You should replace the MATLAB code template, and define approriate inputs and outputs.

4.3.1 Example 1

```
    from nipype.interfaces.matlab import MatlabCommand
    from nipype.interfaces.base import TraitedSpec, BaseInterface, BaseInterfaceInputSpec, File
    import os
    from string import Template

    class ConmapTxt2MatInputSpec(BaseInterfaceInputSpec):
        in_file = File(exists=True, mandatory=True)
        out_file = File('cmatrix.mat', usedefault=True)

    class ConmapTxt2MatOutputSpec(TraitedSpec):
        out_file = File(exists=True)

    class ConmapTxt2Mat(BaseInterface):
        input_spec = ConmapTxt2MatInputSpec
        output_spec = ConmapTxt2MatOutputSpec

        def _run_interface(self, runtime):
            d = dict(in_file=self.inputs.in_file,
            out_file=self.inputs.out_file)
            #this is your MATLAB code template
            script = Template("""in_file = '$in_file';
    out_file = '$out_file';
    ConmapTxt2Mat(in_file, out_file);
    exit;
    """).substitute(d)

            # mfile = True  will create an .m file with your script and executed.
            # Alternatively
            # mfile can be set to False which will cause the matlab code to be
            # passed
            # as a commandline argument to the matlab executable
            # (without creating any files).
            # This, however, is less reliable and harder to debug
            # (code will be reduced to
            # a single line and stripped of any comments).

            mlab = MatlabCommand(script=script, mfile=True)
            result = mlab.run()
            return result.runtime

        def _list_outputs(self):
            outputs = self._outputs().get()
            outputs['out_file'] = os.path.abspath(self.inputs.out_file)
            return outputs
```

4.3.2 Example 2

By subclassing **MatlabCommand** for your main class, and **MatlabInputSpec** for your input and output spec, you gain access to some useful MATLAB hooks

```
import os
from nipype.interfaces.base import File, traits
from nipype.interfaces.matlab import MatlabCommand, MatlabInputSpec


class HelloWorldInputSpec( MatlabInputSpec):
    name = traits.Str( mandatory = True,
                       desc = 'Name of person to say hello to')

class HelloWorldOutputSpec( MatlabInputSpec):
    matlab_output = traits.Str( )

class HelloWorld( MatlabCommand):
    """ Basic Hello World that displays Hello <name> in MATLAB

    Returns
    -------

    matlab_output : capture of matlab output which may be
                    parsed by user to get computation results

    Examples
    --------

    >>> hello = HelloWorld()
    >>> hello.inputs.name = 'hello_world'
    >>> out = hello.run()
    >>> print out.outputs.matlab_output
    """
    input_spec = HelloWorldInputSpec
    output_spec = HelloWorldOutputSpec

    def _my_script(self):
        """This is where you implement your script"""
        script = """
        disp('Hello %s Python')
        two = 1 + 1
        """%(self.inputs.name)
        return script


    def run(self, **inputs):
        ## inject your script
        self.inputs.script =  self._my_script()
        results = super(MatlabCommand, self).run( **inputs)
        stdout = results.runtime.stdout
        # attach stdout to outputs to access matlab results
        results.outputs.matlab_output = stdout
        return results


    def _list_outputs(self):
        outputs = self._outputs().get()
        return outputs
```

4.4 How to wrap a Python script

This is a minimal pure python interface. As you can see all you need to do is to do is to define inputs, outputs, _run_interface() (not run()), and _list_outputs.

```
from nipype.interfaces.base import BaseInterface, \
    BaseInterfaceInputSpec, traits, File, TraitedSpec
from nipype.utils.filemanip import split_filename

import nibabel as nb
import numpy as np
import os

class SimpleThresholdInputSpec(BaseInterfaceInputSpec):
    volume = File(exists=True, desc='volume to be thresholded', mandatory=True)
    threshold = traits.Float(desc='everything below this value will be set to zero',
                             mandatory=True)


class SimpleThresholdOutputSpec(TraitedSpec):
    thresholded_volume = File(exists=True, desc="thresholded volume")


class SimpleThreshold(BaseInterface):
    input_spec = SimpleThresholdInputSpec
    output_spec = SimpleThresholdOutputSpec

    def _run_interface(self, runtime):
        fname = self.inputs.volume
        img = nb.load(fname)
        data = np.array(img.get_data())

        active_map = data > self.inputs.threshold

        thresholded_map = np.zeros(data.shape)
        thresholded_map[active_map] = data[active_map]

        new_img = nb.Nifti1Image(thresholded_map, img.get_affine(), img.get_header())
        _, base, _ = split_filename(fname)
        nb.save(new_img, base + '_thresholded.nii')

        return runtime

    def _list_outputs(self):
        outputs = self._outputs().get()
        fname = self.inputs.volume
        _, base, _ = split_filename(fname)
        outputs["thresholded_volume"] = os.path.abspath(base + '_thresholded.nii')
        return outputs
```

4.5 Working with *nipype* source code

Contents:

4.5.1 Introduction

These pages describe a git and github workflow for the nipype project.

There are several different workflows here, for different ways of working with *nipype*.
This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.
For general resources for learning git see *git resources*.

4.5.2 Install git

Overview

Debian / Ubuntu	`sudo apt-get install git-core`
Fedora	`sudo yum install git-core`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the git page for the most recent information.
Have a look at the github install help pages available from github help
There are good instructions here: http://book.git-scm.com/2_installing_git.html

4.5.3 Following the latest source

These are the instructions if you just want to follow the latest *nipype* source, but you don't need to do any development for now.
The steps are:

- *Install git*
- get local copy of the git repository from github
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/nipy/nipype.git
```

You now have a copy of the code tree in the new `nipype` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd nipype
git pull
```

The tree in `nipype` will now have the latest changes from the initial repository.

4.5.4 Making a patch

You've discovered a bug or something else you want to change in nipype .. — excellent!
You've worked out a way to fix it — even better!
You want to tell us about it — best of all!
The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/nipy/nipype.git
# make a branch for your patching
cd nipype
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the nipype mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the nipype repository:

```
git clone git://github.com/nipy/nipype.git
cd nipype
```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the nipype mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the nipype repository on github — *Making your own copy (fork) of nipype*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/nipype.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

4.5.5 Git for development

Contents:

Making your own copy (fork) of nipype

You need to do this only once. The instructions here are very similar to the instructions at http://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the nipype project, and to suggest some default names.

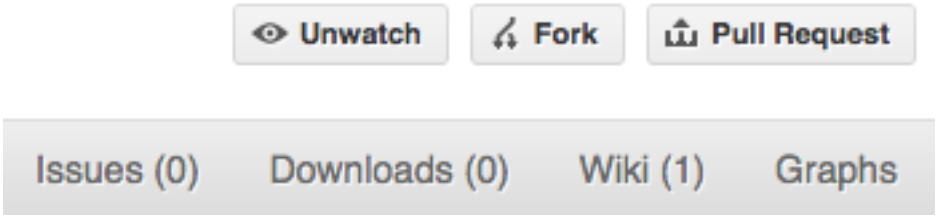
Set up and configure a github account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of nipype

1. Log into your github account.
2. Go to the nipype github home at nipype github.
3. Click on the *fork* button:



Now, after a short pause and some 'Hardcore forking action', you should find yourself at the home page for your own forked copy of nipype.

Set up your fork

First you follow the instructions for *Making your own copy (fork) of nipype*.

Overview

```
git clone git@github.com:your-user-name/nipype.git
cd nipype
git remote add upstream git://github.com/nipy/nipype.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/nipype.git`
2. Investigate. Change directory to your new repo: `cd nipype`. Then `git branch -a` to show you all branches. You'll get something like:

```
* master
remotes/origin/master
```

This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.
Now you want to connect to the upstream nipype github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd nipype
git remote add upstream git://github.com/nipy/nipype.git
```

`upstream` here is just the arbitrary name we're using to refer to the main nipype repository at nipype github.
Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.
Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/nipy/nipype.git (fetch)
upstream     git://github.com/nipy/nipype.git (push)
origin       git@github.com:your-user-name/nipype.git (fetch)
origin       git@github.com:your-user-name/nipype.git (push)
```

Configure git

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory. Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com
```

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases You might well benefit from some aliases to common commands.
For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`
The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Development workflow

You already have your own forked copy of the nipype repository, by following *Making your own copy (fork) of nipype*, *Set up your fork*, and you have configured git by following *Configure git*.

Workflow summary

- Keep your `master` branch clean of edits that have not been merged to the main nipype development repo. Your `master` then will follow the main nipype repository.
- Start a new *feature branch* for each set of edits that you do.
- If you can avoid it, try not to merge other branches into your feature branch while you are working.
- Ask for review!
  This way of working really helps to keep work well organized, and in keeping history as clear as possible.
  See — for example — linux git workflow.

Making a new feature branch

```
git branch my-new-feature
git checkout my-new-feature
```

Generally, you will want to keep this also on your public github fork of nipype. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

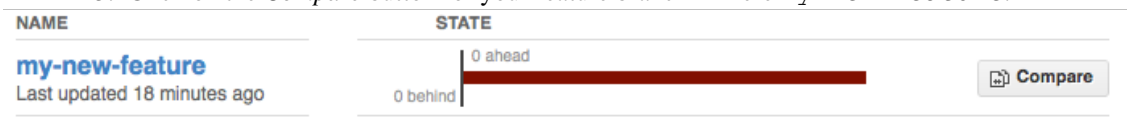
```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see *git push*).

Asking for code review

1. Go to your repo URL — e.g. `http://github.com/your-user-name/nipype`.
2. Click on the *Branch list* button:

3. Click on the *Compare* button for your feature branch — here `my-new-feature`:



4. If asked, select the *base* and *comparison* branch names you want to compare. Usually these will be `master` and `my-new-feature` (where that is your feature branch name).
5. At this point you should get a nice summary of the changes. Copy the URL for this, and post it to the nipype mailing list, asking for review. The URL will look something like: `http://github.com/your-user-name/nipype/compare/master...my-new-feature`. There's an example at http://github.com/matthew-brett/nipy/compare/master...find-install-data See: http://github.com/blog/612-introducing-github-compare-view for more detail.

The generated comparison, is between your feature branch `my-new-feature`, and the place in `master`

from which you branched `my-new-feature`. In other words, you can keep updating `master` without interfering with the output from the comparison. More detail? Note the three dots in the URL above (`master...my-new-feature`).

Two vs three dots
Imagine a series of commits A, B, C, D... Imagine that there are two branches, *topic* and *master*. You branched *topic* off *master* when *master* was at commit 'E'. The graph of the commits looks like this:

```
      A---B---C topic
      /
D---E---F---G master
```

Then:

```
git diff master..topic
```

will output the difference from G to C (i.e. with effects of F and G), while:

```
git diff master...topic
```

would output just differences in the topic branch (i.e. only A, B, and C). [1]

Asking for your changes to be merged with the main repo

When you are ready to ask for the merge of your code:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/nipype.git`.
2. Click on the 'Pull request' button:

Enter a message; we suggest you select only `nipype` as the recipient. The message will go to the nipype mailing list. Please feel free to add others from the list as you like.

Merging from trunk

This updates your code from the upstream nipype github repo.

Overview

```
# go to your master branch
git checkout master
# pull changes from github
git fetch upstream
# merge from upstream
git merge upstream/master
```

In detail We suggest that you do this only for your `master` branch, and leave your 'feature' branches unmerged, to keep their history as clean as possible. This makes code review easier:

```
git checkout master
```

[1] Thanks to Yarik Halchenko for this explanation.

Make sure you have done *Linking your repository to the upstream repo*.
Merge the upstream code into your current development by first pulling the upstream repo to a copy on your local machine:

```
git fetch upstream
```

then merging into your current branch:

```
git merge upstream/master
```

Deleting a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

(Note the colon `:` before `test-branch`. See also: http://github.com/guides/remove-a-remote-branch

Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.
First fork nipype into your account, as from *Making your own copy (fork) of nipype*.
Then, go to your forked repository github page, say `http://github.com/your-user-name/nipype`
Click on the 'Admin' button, and add anyone else to the repo as a collaborator:

Now all those people can do:

```
git clone git@githhub.com:your-user-name/nipype.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.
Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Exploring your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

4.5.6 git resources

Tutorials and summaries

- github help has an excellent series of how-to guides.
- learn.github has an excellent series of tutorials
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- The git community book
- git ready — a nice series of tutorials
- git casts — video snippets giving git how-tos.
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- Our own git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

4.6 Architecture (discussions from 2009)

This section reflects notes and discussion between developers during the start of the nipype project in 2009.

4.6.1 Design Guidelines

These are guidelines that the core nipype developers have agreed on:
Interfaces should keep all parameters affecting construction of the appropriate command in the "input" bunch.
The .run() method of an Interface should include all required inputs as explicitly named parameters, and they should take a default value of None.

Any Interface should at a minimum support cwd as a command-line argument to .run(). This may be accomplished by allowing cwd as an element of the input Bunch, or handled as a separate case.
Relatedly, any Interface should output all files to cwd if it is set, and otherwise to os.getcwd() (or equivalent).
We need to decide on a consistent policy towards the maintinence of paths to files. It seems like the best strategy might be to do absolute (os.realpath?) filenames by default, allowing for relative paths by explicitly including something that doesn't start with a '/'. This could include '.' in some sort of path-spec.
Class attributes should never be modified by an instance of that class. And probably not ever.

4.6.2 Providing for Provenance

The following is a specific discussion that should be thought out an more generally applied to the way we handle auto-generation / or "sourcing" of settings in an interface.
There are two possible sources (at a minimum) from which the interface instance could obtain "outputtype" - itself, or FSLInfo. Currently, the outputtype gets read from FSLInfo if self.outputtype (er, _outputtype?) is None.
In the case of other opt_map specifications, there are defaults that get specified if the value is None. For example output filenames are often auto-generated. If you look at the code for fsl.Bet for example, there is no way for the outfile to get picked up at the pipeline level, because it is a transient variable. This is OK, as the generation of the outfile name is contingent ONLY on inputs which ARE available to the pipeline machinery (i.e., via inspection of the Bet instance's attributes).
However, with outputtype, we are in a situation in which "autogeneration" incorporates potentially transient information external to the instance itself. Thus, some care needs to be taken in always ensuring this information is hashable.

4.6.3 Design Principles

These are (currently) Dav Clark's best guess at what the group might agree on:
It should be very easy to figure out what was done by the pypeline.
Code should support relocatability - this could be via URIs, relative paths or potentially other mechanisms.
Unless otherwise called for, code should be thread safe, just in case.
The pipeline should make it easy to change aspects of an analysis with minimal recomputation, downloading, etc. (This is not the case currently - any change will overwrite the old node). Also, the fact that multiple files get rolled into a single node is problematic for similar reasons. E.g. - node([file1 ... file100]) will get recomputed if we add only one file!.
However, it should also be easy to identify and delete things you don't need anymore.
Pipelines and bits of pipelines should be easy to share.
Things that are the same should be called the same thing in most places. For interfaces that have an obvious meaning for the terms, "infiles" and "outfile(s)". If a file is in both the inputs and outputs, it should be called the same thing in both places. If it is produced by one interface and consumed by another, same thing should be used.

4.6.4 Discussions

Auto-generated filenames

In refactoring the inputs in the traitlets branch I'm working through the different ways that filenames are generated and want to make sure the interface is consistent. The notes below are all using fsl.Bet as that's the first class we're Traiting. Other interface classes may handle this differently, but should agree on a convention and apply it across all Interfaces (if possible).

Current Rules

These rules are for fsl.Bet, but it appears they are the same for all fsl and spm Interfaces.
Bet has two mandatory parameters, `infile` and `outfile`. These are the rules for how they are handled in different use cases.

1. If `infile` or `outfile` are absolute paths, they are used as-is and never changed. This allows users to override any filename/path generation.
2. If `outfile` is not specified, a filename is generated.
3. Generated filenames (at least for `outfile`) are based on:

- `infile`, the filename minus the extensions.
- A suffix specified by the Interface. For example Bet uses *_brain* suffix.
- The current working directory, os.getcwd(). Example:
  If `infile` == 'foo.nii' and the cwd is `/home/cburns` then generated `outfile` for Bet will be `/home/cburns/foo_brain.nii.gz`

4. If `outfile` is not an absolute path, for instance just a filename, the absolute path is generated using `os.path.realpath`. This absolute path is needed to make sure the packages (Bet in this case) write the output file to a location of our choosing. The generated absolute path is only used in the `cmdline` at runtime and does __not__ overwrite the class attr `self.inputs.outfile`. It is generated only when the `cmdline` is invoked.

Walking through some examples

In this example we assign `infile` directly but `outfile` is generated in `Bet._parse_inputs` based on `infile`. The generated `outfile` is only used in the cmdline at runtime and not stored in `self.inputs.outfile`. This seems correct.

```
In [15]: from nipype.interfaces import fsl

In [16]: mybet = fsl.Bet()

In [17]: mybet.inputs.infile = 'foo.nii'

In [18]: res = mybet.run()

In [19]: res.runtime.cmdline
Out[19]: 'bet foo.nii /Users/cburns/src/nipy-sf/nipype/trunk/nipype/interfaces/tests/foo_brain.n

In [21]: mybet.inputs
Out[21]: Bunch(center=None, flags=None, frac=None, functional=None,
infile='foo.nii', mask=None, mesh=None, nooutput=None, outfile=None,
outline=None, radius=None, reduce_bias=None, skull=None, threshold=None,
verbose=None, vertical_gradient=None)

In [24]: mybet.cmdline
Out[24]: 'bet foo.nii /Users/cburns/src/nipy-sf/nipype/trunk/nipype/interfaces/tests/foo_brain.n

In [25]: mybet.inputs.outfile

In [26]: mybet.inputs.infile
Out[26]: 'foo.nii'
```

We get the same behavior here when we assign `infile` at initialization:

```
In [28]: mybet = fsl.Bet(infile='foo.nii')

In [29]: mybet.cmdline
Out[29]: 'bet foo.nii /Users/cburns/src/nipy-sf/nipype/trunk/nipype/interfaces/tests/foo_brain.n

In [30]: mybet.inputs
Out[30]: Bunch(center=None, flags=None, frac=None, functional=None,
infile='foo.nii', mask=None, mesh=None, nooutput=None, outfile=None,
outline=None, radius=None, reduce_bias=None, skull=None, threshold=None,
verbose=None, vertical_gradient=None)
```

```
In [31]: res = mybet.run()

In [32]: res.runtime.cmdline
Out[32]: 'bet foo.nii /Users/cburns/src/nipy-sf/nipype/trunk/nipype/interfaces/tests/foo_brain.n
```

Here we specify absolute paths for both `infile` and `outfile`. The command line's look as expected:

```
In [53]: import os

In [54]: mybet = fsl.Bet()

In [55]: mybet.inputs.infile = os.path.join('/Users/cburns/tmp/junk', 'foo.nii')
In [56]: mybet.inputs.outfile = os.path.join('/Users/cburns/tmp/junk', 'bar.nii')

In [57]: mybet.cmdline
Out[57]: 'bet /Users/cburns/tmp/junk/foo.nii /Users/cburns/tmp/junk/bar.nii'

In [58]: res = mybet.run()

In [59]: res.runtime.cmdline
Out[59]: 'bet /Users/cburns/tmp/junk/foo.nii /Users/cburns/tmp/junk/bar.nii'
```

Here passing in a new `outfile` in the `run` method will update `mybet.inputs.outfile` to the passed in value. Should this be the case?

```
In [110]: mybet = fsl.Bet(infile='foo.nii', outfile='bar.nii')

In [111]: mybet.inputs.outfile
Out[111]: 'bar.nii'

In [112]: mybet.cmdline
Out[112]: 'bet foo.nii /Users/cburns/src/nipy-sf/nipype/trunk/nipype/interfaces/tests/bar.nii'

In [113]: res = mybet.run(outfile = os.path.join('/Users/cburns/tmp/junk', 'not_bar.nii'))

In [114]: mybet.inputs.outfile
Out[114]: '/Users/cburns/tmp/junk/not_bar.nii'

In [115]: mybet.cmdline
Out[115]: 'bet foo.nii /Users/cburns/tmp/junk/not_bar.nii'
```

In this case we provide `outfile` but not as an absolue path, so the absolue path is generated and used for the `cmdline` when run, but `mybet.inputs.outfile` is not updated with the absolute path.

```
In [74]: mybet = fsl.Bet(infile='foo.nii', outfile='bar.nii')

In [75]: mybet.inputs.outfile
Out[75]: 'bar.nii'

In [76]: mybet.cmdline
Out[76]: 'bet foo.nii /Users/cburns/src/nipy-sf/nipype/trunk/nipype/interfaces/tests/bar.nii'

In [77]: res = mybet.run()

In [78]: res.runtime.cmdline
Out[78]: 'bet foo.nii /Users/cburns/src/nipy-sf/nipype/trunk/nipype/interfaces/tests/bar.nii'

In [80]: res.interface.inputs.outfile
Out[80]: 'bar.nii'
```

4.7 W3C PROV support

4.7.1 Overview

We're using the the W3C PROV data model to capture and represent provenance in Nipype.
For an overview see:
PROV-DM overview
Each interface writes out a provenance.json (currently prov-json) or provenance.rdf (if rdflib is available) file. The workflow engine can also write out a provenance of the workflow if instructed.
This is very much an experimental feature as we continue to refine how exactly the provenance should be stored and how such information can be used for reporting or reconstituting workflows. By default provenance writing is disabled for the 0.9 release, to enable insert the following code at the top of your script:

```
>>> from nipype import config
>>> config.enable_provenance()
```

4.8 Software using Nipype

4.8.1 Configurable Pipeline for the Analysis of Connectomes (C-PAC)

C-PAC is an open-source software pipeline for automated preprocessing and analysis of resting-state fMRI data. C-PAC builds upon a robust set of existing software packages including AFNI, FSL, and ANTS, and makes it easy for both novice users and experts to explore their data using a wide array of analytic tools. Users define analysis pipelines by specifying a combination of preprocessing options and analyses to be run on an arbitrary number of subjects. Results can then be compared across groups using the integrated group statistics feature. C-PAC makes extensive use of Nipype Workflows and Interfaces.

4.8.2 BRAINSTools

BRAINSTools is a suite of tools for medical image processing focused on brain analysis.

4.8.3 Brain Imaging Pipelines (BIPs)

BIPs is a set of predefined Nipype workflows coupled with a graphical interface and ability to save and share workflow configurations. It provides both Nipype Workflows and Interfaces.

4.8.4 BROCCOLI

BROCCOLI is a piece of software for fast fMRI analysis on many core CPUs and GPUs. It provides Nipype Interfaces.

4.8.5 Forward

Forward is set of tools simplifying the preparation of accurate electromagnetic head models for EEG forward modeling. It uses Nipype Workflows and Interfaces.

4.8.6 Limbo

Limbo is a toolbox for finding brain regions that are neither significantly active nor inactive, but rather "in limbo". It was build using custom Nipype Interfaces and Workflows.

4.8.7 Lyman

Lyman is a high-level ecosystem for analyzing task based fMRI neuroimaging data using open-source software. It aims to support an analysis workflow that is powerful, flexible, and reproducible, while automating as much

of the processing as possible. It is build upon Nipype Workflows and Interfaces.

4.8.8 Medimsight

Medimsight is a commercial service medical imaging cloud platform. It uses Nipype to interface with various neuroimaging software.

4.8.9 MIA

MIA MIA is a a toolkit for gray scale medical image analysis. It provides Nipype interfaces for easy integration with other software.

4.8.10 Mindboggle

Mindboggle software package automates shape analysis of anatomical labels and features extracted from human brain MR image data. Mindboggle can be run as a single command, and can be easily installed as a cross-platform virtual machine for convenience and reproducibility of results. Behind the scenes, open source Python and C++ code run within a Nipype pipeline framework.

4.8.11 OpenfMRI

OpenfMRI is a repository for task based fMRI datasets. It uses Nipype for automated analysis of the deposited data.

4.8.12 serial functional Diffusion Mapping (sfDM)

'sfDM <http://github.com/PIRCImagingTools/sfDM>'_ is a software package for looking at changes in diffusion profiles of different tissue types across time. It uses Nipype to process the data.

4.8.13 The Stanford CNI MRS Library (SMAL)

SMAL is a library providing algorithms and methods to read and analyze data from Magnetic Resonance Spectroscopy (MRS) experiments. It provides an API for fitting models of the spectral line-widths of several different molecular species, and quantify their relative abundance in human brain tissue. SMAL uses Nipype Workflows and Interfaces.

4.8.14 tract_querier

tract_querier is a White Matter Query Language tool. It provides Nipype interfaces.

Interfaces, Workflows and Examples

- Workflows
- Examples
- Interfaces

Symbols

C

I

M

N

P

W